END

DATE
FILMED

3—77

ADA035719

REPORT DOCUMENTATION PAGE

1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
Memo No. 380 ⑭ AI-M-380		

4. TITLE (and Subtitle)	5. TYPE OF REPORT & PERIOD COVERED
Forward Reasoning and Dependency-Directed Backtracking In a System for Computer-Aided Circuit Analysis.	Memo
	6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)	8. CONTRACT OR GRANT NUMBER(s)
Richard M. Stallman and Gerald Jay Sussman	N00014-75-C-0643

9. PERFORMING ORGANIZATION NAME AND ADDRESS	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139	⑫ 74 p.

11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE
Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209	Sept 1976
	13. NUMBER OF PAGES
	69

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report)
Office of Naval Research Information Systems Arlington, Virginia 22217	UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Artificial Intelligence	Backtracking
Computer Aided Circuit Analysis	Dependency Directed Backtracking
Pattern-Directed Invocation	Deduction
Control of Reasoning	Rule-based Systems
Propagation of Constraints	Search

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We present a rule-based system for computer-aided circuit analysis. The set of rules, called EL, is written in a rule language called ARS. Rules are implemented by ARS as pattern-directed invocation demons monitoring an associative data base. Deductions are performed in an antecedent manner, giving EL's analysis a catch-as-catch-can flavor suggestive of the behavior of expert circuit analyzers. We call this style of circuit analysis is called propogation of constraints. The system threads deduced facts with →OVER

BLOCK 20 CONTINUED

justifications which mention the antecedent facts and the rule used.
These justifications may be examined by the user to gain insight into the
operation of the set of rules as they apply to a problem. The same justifica-
tions are used by the system to determine the currently active data-base context
for reasoning in hypothetical situations. They are also used by the system
in the analysis failures to reduce the search space. This leads to effective
control of combinatorial search which we call <u>dependency-directed backtracking.</u>

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Memo No. 380   Sept. 1976

Forward Reasoning and Dependency-Directed Backtracking In a System for Computer-Aided Circuit Analysis

by Richard M. Stallman and Gerald Jay Sussman

Abstract:

We present a rule-based system for computer-aided circuit analysis. The set of rules, called EL, is written in a rule language called ARS. Rules are implemented by ARS as pattern-directed invocation demons monitoring an associative data base. Deductions are performed in an antecedent manner, giving EL's analysis a catch-as-catch-can flavor suggestive of the behavior of expert circuit analyzers. We call this style of circuit analysis <u>propagation of constraints</u>. The system threads deduced facts with justifications which mention the antecedent facts and the rule used. These justifications may be examined by the user to gain insight into the operation of the set of rules as they apply to a problem. The same justifications are used by the system to determine the currently active data-base context for reasoning in hypothetical situations. They are also used by the system in the analysis failures to reduce the search space. This leads to effective control of combinatorial search which we call <u>dependency-directed backtracking</u>.

Contents:

Introduction

A major problem confronting builders of automatic problem-solving systems is that of the combinatorial explosion of search-spaces. One way to attack this problem is to build systems that effectively use the results of failures to reduce the search space -- that learn from their exploration of blind alleys.[Blind alleys] Another way is to represent the problems and their solutions in such a way that combinatorial searches are self limiting.[Limiters]

A second major problem is the difficulty of debugging programs containing large amounts of knowledge. The complexity of the interactions between the "chunks" of knowledge makes it difficult to ascertain what is to blame when a bug manifests itself.[Complexity] One approach to this problem is to build systems which remember and explain their reasoning.[Explainers] Such programs are more convincing when right, and easier to debug when wrong.

We have designed and implemented[LISP] a problem-solving language called ARS[ARS] in which problem-solving rules are represented as demons with multiple patterns of invocation[Pattern-directed invocation] monitoring an associative data base.[Data bases] It performs all deductions in an antecedent manner, threading the deduced facts with justifications which mention the antecedent facts used and the rule of inference applied. These justifications may be examined by the user to gain insight into the operation of the system of rules as they apply to a problem. The same justifications are employed by the system to determine the currently active data-base context for reasoning in hypothetical situations.[Context] Justifications are also used in the analysis of blind alleys to extract information which will limit future search.

We have used ARS to implement a set of rules for electronic circuit analysis. This set of rules, a version of EL,[EL] encodes familiar approximations to physical laws such as Kirchoff's laws and Ohm's law as well as models for more complex devices such as transistors. Facts, which may be given or deduced, represent data such as the circuit topology, device parameters, and voltages and currents. The antecedent reasoning of ARS gives analysis by EL a "catch-as-catch-can" flavor suggestive of the behavior of a circuit expert. The justifications prepared by ARS allow an EL user to examine the basis of its conclusions. This is useful in understanding the operation of the circuit as well as in debugging the EL rules. For example, a device parameter not mentioned in the derivation of a voltage value has no part in determining that value. If a user changes some part of the circuit specification (a device parameter or an imposed voltage or current), only those facts depending on the changed fact need be "forgotten" and re-deduced, so small changes in the circuit may need only a small amount of new analysis. Finally, the search-limiting combinatorial methods supplied by ARS lead to efficient analysis of circuits with piecewise-linear models.

The application of a rule in ARS implements a <u>one-step deduction</u>. A few examples of one-step deductions, resulting from the application of some EL rules in the domain of resistive network analysis, are:

1: If the voltage on one terminal of a voltage source is given, one can assign the voltage on the other terminal.

2: If the voltage on both terminals of a resistor are given, and the resistance is known, then the current through it can be assigned.

3: If the current through a resistor, and the voltage on one of its terminals, is known, along

with the resistance of the resistor, then the voltage on the other terminal can be assigned.
4: If all but one of the currents into a node are given, the remaining current can be assigned.

The style of analysis performed by EL, which we call the method of propagation of constraints,[Propagation] requires the introduction and manipulation of some symbolic quantities. Though the system has routines for symbolic algebra,[Symbolic manipulation] they can handle only linear relationships. Nonlinear devices such as transistors are represented by piecewise-linear models that cannot be used symbolically; they can be applied only after one has guessed[Advice] a particular operating region for each nonlinear device in the circuit. Trial and error can find the right regions but this method of assumed states is potentially combinatorially explosive. ARS supplies dependency-directed backtracking, a scheme which limits the search as follows: The system notes a contradiction when it attempts to solve an impossible algebraic relationship, or when discovers that a transistor's operating point is not within the possible range for its assumed region. The antecedents of the contradictory facts are scanned to find which nonlinear device state guesses (more generally, the backtrackable choicepoints) are relevant; ARS never tries that combination of guesses again. A short list of relevant choicepoints eliminates from consideration a large number of combinations of answers to all the other (irrelevant) choices. This is how the justifications (or dependency records) are used to extract and retain more information from each contradiction than a chronological backtracking system.[Backtracking] A chronological backtracking system would often have to try many more combinations, each time wasting much labor rediscovering the original contradiction.

How it works:

In EL all circuit-specific knowledge is represented as assertions in a relational data base. General knowledge about circuits is represented by laws, which are demons subject to pattern-directed invocation. Some laws represent knowledge as equalities. For example, there is one demon for Ohm's law for resistors, one demon that knows that the current going into one terminal of a resistor must come out of the other, one demon that knows that the currents on the wires coming into a node must sum to zero, etc. Other laws, called Monitors handle knowledge in the form of inequalities: For example, I-MONITOR-DIODE knows that a diode can have a forward current if and only if it is ON, and can never have a backward current.

When an assertion (for example, (= (VOLTAGE (C Q1)) 3.4), which says that the voltage on Q1's collector has the value 3.4 volts) is added to the data base, several demons will in general match it and be triggered. (In this example, they will include DC-KVL, which makes sure that all other elements' terminals connected to Q1's collector are also known to have that voltage, and VCE-MONITOR-BJT, which checks that Q1 is correctly biased for its assumed operating region.). The names of the triggered laws are put on a queue, together with arguments such as the place in the circuit that the law is to operate. Eventually they will be taken off the queue and processed, perhaps making new deductions and starting the cycle over again.

When a law is finally processed, it can do two useful things: make a new assertion (or several), or detect a contradiction. A new assertion is entered in the data base and has its antecedents recorded; they are the asserting demon itself, and all the assertions which invoked it or were used by it. This complete memory of how every datum was deduced becomes useful when a

contradiction is to be handled. A contradiction indicates that some previously made arbitrary choice (e.g. an assumption of the linear operating region of some nonlinear component) was incorrect. ARS scans backward along the chains of deduction from the scene of the contradiction, to find those choices which contributed to the contradiction, and records them all in a NOGOOD assertion to make sure that the same combination is never tried again. (NOGOOD ((MODE Q1) CUTOFF) ((MODE D5) ON)) is a NOGOOD assertion that says that it cannot be simultaneously true that transistor Q1 is cut off and diode D5 is conducting. Such a NOGOOD might be deduced if Q1 and D5 were connected in series. Next, one of the conspiring choices is arbitrarily called the "culprit" ("scape-goat" might be a better term) and re-chosen differently. This is not mere undirected trial and error search as occurs when chronological backtracking with a sequential control structure is used, since it is guaranteed not to waste time trying alternative answers to an irrelevant question. The NOGOOD assertion is a further innovation that saves even more computation by reducing the size of the search space, since it contains not *all* the choices in effect, but only those that were *specifically used* in deducing the contradiction. Frequently some of the circuit's transistors will not be mentioned at all. Then, the NOGOOD applies regardless of the states assumed for those irrelevant transistors. If there are ten transistors in the circuit not mentioned in the NOGOOD, then since every transistor has three states (in the EL model) the single NOGOOD has ruled out 3^{10}=59049 different states of the whole circuit.

Analysis by Propagation of Constraints

Consider a simple voltage divider:
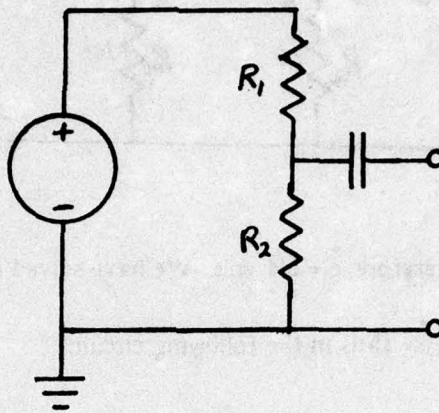
Suppose that the voltage at the midpoint is known to be 3 volts, relative to the indicated ground. Since there is known to be no DC current through the capacitor, it is possible to determine the strength of the voltage source. Forward reasoning is doing it this way: First, use Ohm's law to compute the current through R_2 from its resistance and the difference of the voltages on its terminals. Next, the current through R_1 can be seen, via KCL, to be the same as that through R_2. Finally, that current, together with R_1's resistance and the voltage at the midpoint, can be fed to Ohm's law to produce the voltage at the top. This is an example of what we call "forward reasoning" or (as applied to circuits) "propagation of constraints".

However, not all circuit problems can be solved so simply. Consider a ladder network:
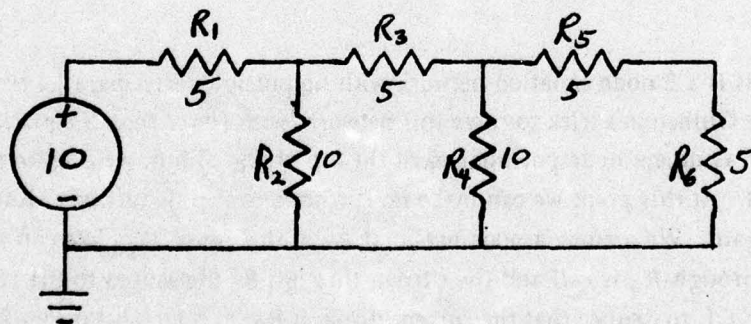
Such a network might be solved with three node equations or by series-parallel reduction. More in the spirit of forward reasoning is "Guillemin's Trick":

We assume a node voltage, e, at the end of the ladder. This implies a current, e/5, going down R_6 by Ohm's Law. KCL then tells us that this current must come out of R_5. But we know the voltage on the right of R_5 and the current through it, so we deduce that the voltage on its left is 2e. We use this voltage to deduce the current through R_4 and then KCL to give us the current through R_3. We continue this process until we get all node voltages defined in terms of e:
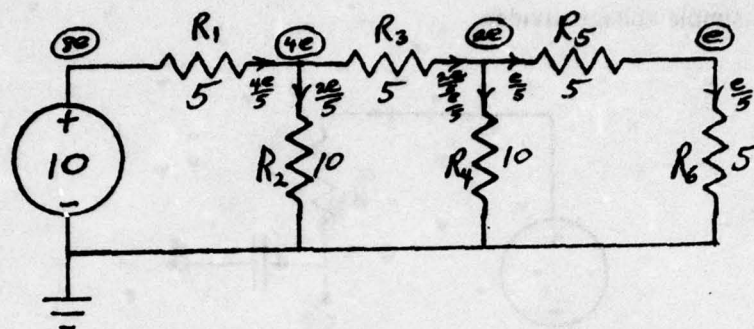
But now we know $8e = 10$, therefore, $e = 5/4$ volt. We have solved the network with 1 equation in 1 unknown.

Alas, Guillemin's trick fails in the following circuit:
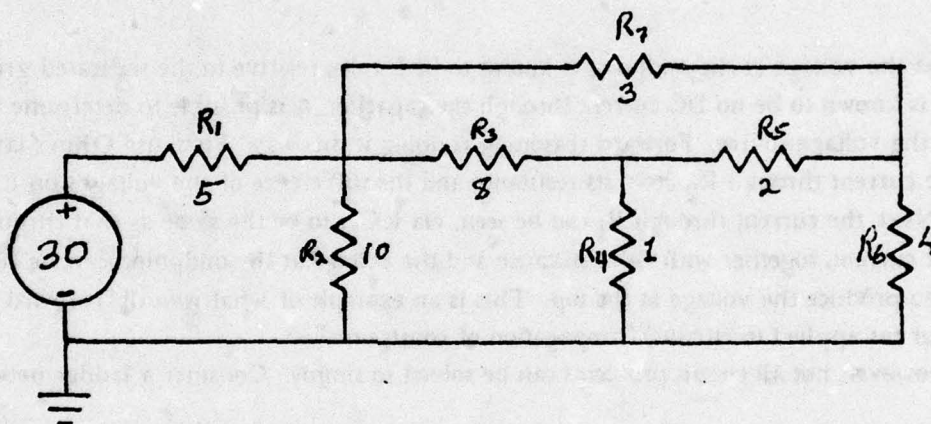
At first glance, this is a 3 node equation network with no possible series-parallel reductions. We want to generalize Guillemin's trick to solve this network (with fewer than 3 equations).

We first assume a node potential, e_1, at the top of R_6. Thus, we deduce that the current through R_6 is $e_1/4$. At this point we can make no further one-step deductions. Rather than give up, let's poke it again. We assume a node potential, e_2, at the top of R_4. We can now conclude that the current through R_4 is $e_2/1$ and the current through R_5 (measured to the right) is $(e_2-e_1)/2$. We can now use KCL to deduce that the current through R_7 (to the right) is $(3e_1-2e_2)/4$ and that the current through R_3 is $(3e_2-e_1)/2$:
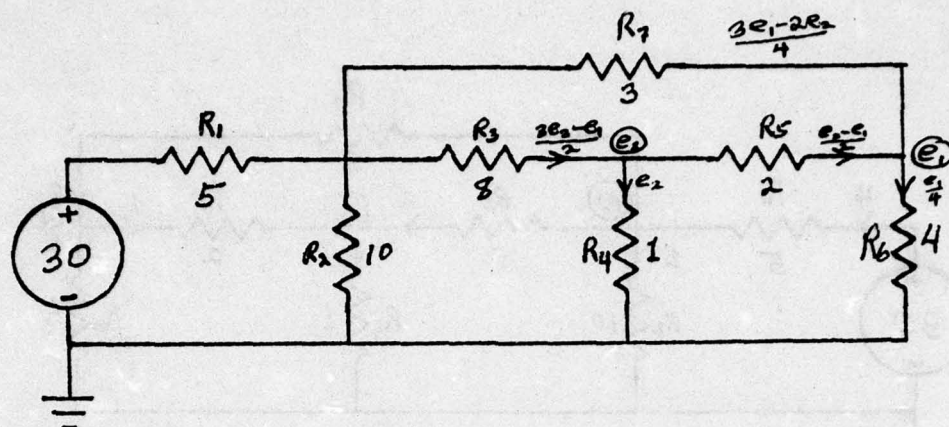
At this point we can deduce that the voltage on the leftmost terminal of R_7 is $e_1+(3/4)(3e_1-2e_2)$ or $(13e_1-6e_2)/4$. We can also deduce that the voltage on the leftmost terminal of R_3 is $e_2+8((3e_2-e_1)/2)=13e_2-4e_1$. Since these terminals are connected together we have two expressions for the same node voltage. Setting them equal and simplifying, we get $e_1=2e_2$. The result of this simplification is:
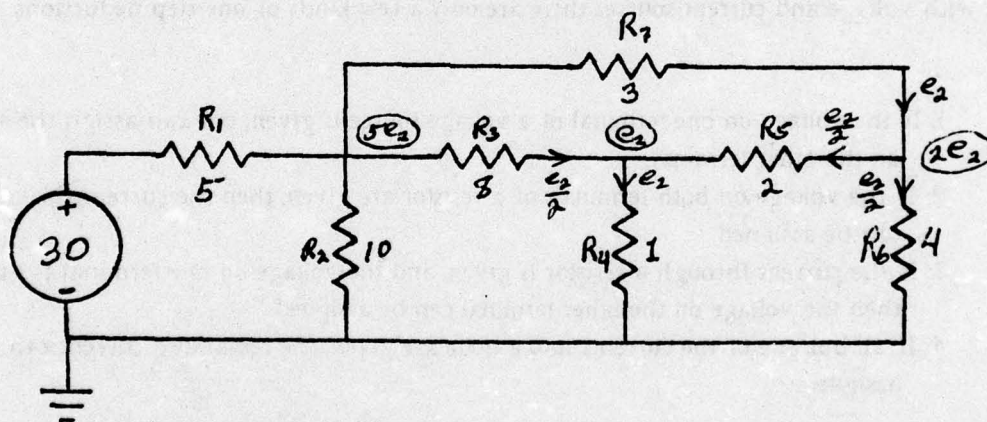
Continuing, we deduce that the current through R_2 must be $5e_2/10=e_2/2$. By KCL, the current through R_1 must be $2e_2$. Ohm's law now gives as the voltage at the left of R_1 as $15e_2$. But this voltage is set by the voltage source, so $15e_2=30$. We conclude that $e_2=2$:
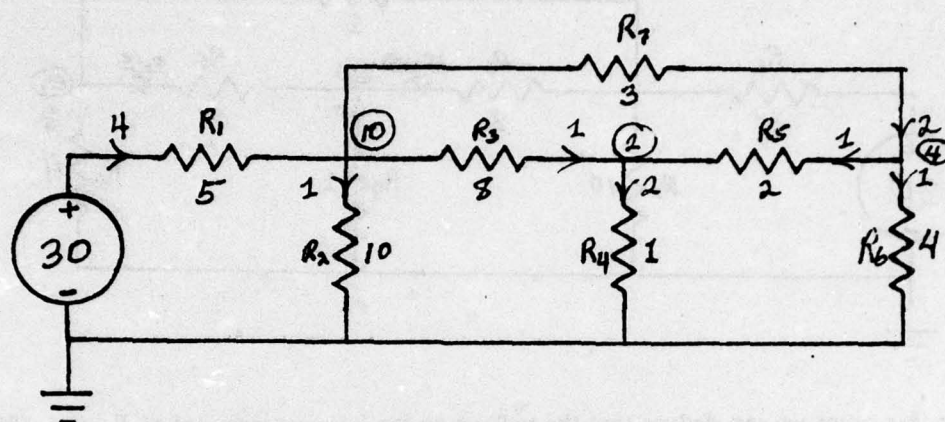
We have solved the network with only two unknowns. As we approach a full graph network, this method degrades smoothly to be the node method, but usually it uses far fewer unknowns.

What have we been doing?

A fundamental concept here is that of a <u>one-step deduction</u>. In the case of a resistive network with voltage and current sources there are only a few kinds of one-step deductions possible:

1: If the voltage on one terminal of a voltage source is given, one can assign the voltage on the other terminal.

2: If the voltage on both terminals of a resistor are given, then the current through it can be assigned.

3: If the current through a resistor is given, and the voltage on one terminal is given, then the voltage on the other terminal can be assigned.

4: If all but one of the currents into a node are given, the remaining current can be assigned.

Another basic concept here is that of a <u>coincidence</u>. A coincidence occurs when a one-step deduction is made which assigns a value to a network variable which already has a value. We have seen several coincidences. In the ladder network example a one-step deduction of type 3 assigns the node voltage 8e to a node which is already at 10 volts. In the second example, the node at the top of R_2 was assigned two different node voltages by two one-step deductions of type 3, and the voltage $15e_2$ even though it already was known to be 30 volts. In each of these cases the coincidence resulted in the formulation of an equation between the competing assignments. At the time of a coincidence, the resulting equation should be solved, if possible, for one of its unknowns in terms of the others. The circuit is then redrawn with that unknown eliminated.

Thus, the basic propagation analysis algorithm is rather simple:

Algorithm: Propagation of Constraints
        Choose a datum node and assign it a potential of 0.
loop: IF there is a one-step deduction available
        Choose a deduction and make it.
                ADVICE:
                    [1] IF the last action was the assignment of
                        a node potential, look for a type 1, 2, or 3
                        deduction involving that node.
                    [2] IF the last action assigned a current,
                        look for a type 3 or 4 deduction involving
                        that branch.
            IF the deduction caused a coincidence THEN
                IF the equation implied by the coincidence is a
                    tautology
                          Ignore the coincidence (and be
                          reassured by the fact that it checks!).
                  contradiction
                      ERROR: You did something wrong.
              otherwise
                Solve for one unknown in terms
                of the others (or for a number, if
                there are no others!). Eliminate
                that unknown throughout the circuit.
        Go to loop.
    IF there is a node without a node potential
        Choose such a node and assign it a new node potential variable.
        Go to loop.
RETURN

All of the unknowns introduced by the algorithm are sure to have had their values determined by the time the algorithm returns.

Now, what about choosing where to place unknowns? We want to get as much action as we can out of each one. One measure of the simultaneity of the situation is given by a count of the number of unknown nodes connected to a given one.[Braida] For example, consider our ladder again. All nodes except the ground and the top of the voltage source are unknown. In the following circuit each unknown node has been annotated with the number of unknowns it is connected to.
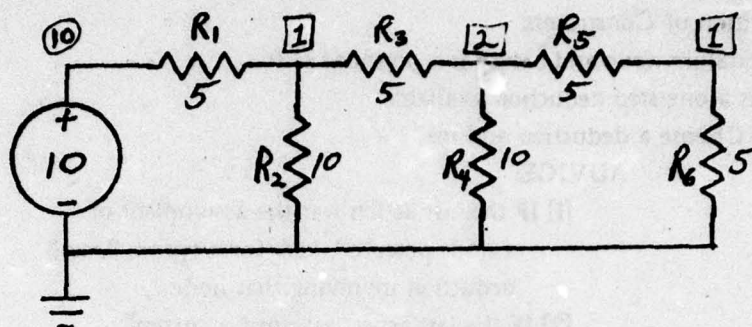
We can see that the middle node has two unknown neighbors while the others have only one. If we place a node potential on the middle node, we can only deduce one current while if we place it at either one of the 1 neighbor nodes we will get the whole answer. The rule is to place the node potential at a node of minimum unknown neighbors. This also has bearing on where to place the datum. In general, we want a node which is maximally connected to unknowns, so as to constrain them quickly.

Facts and Laws

Analysis by propagation of constraints is a form of antecedent reasoning in which a given, closed set of questions (in this case, the voltages and currents at all points in the network) is to be answered. ARS provides a general framework for the implementation of antecedent reasoning systems; we developed it to support the particular set of rules for electrical analysis which we call EL. Like any programming language, ARS is best explained by showing how it can be used to implement an example. We will now display sample parts of EL, and how they implement parts of the analysis method already described.

To EL a circuit is made up of <u>devices</u> and <u>nodes</u>. A device is any of the components one would normally think of as present in the circuit, such as resistors, capacitors, transistors, and voltage sources. Each device has two or more terminals by which it is connected to the rest of the circuit. But two device terminals are never connected *directly*. Instead, they are both connected to a common node (that is the sole purpose of nodes).

ARS requires that all knowlege to be manipulated be represented as assertions, and their manipulators to be expressed as demons. EL therefore deals with facts that name the devices and nodes in the circuit, and state which terminals connect to which nodes. A node or device is named by an <u>IS-A</u> assertion, such as (IS-A R1 RESISTOR) or (IS-A N54 NODE). The IS-A assertions serve several purposes. They control the matches of restricted variables and they enable EL to find all the nodes in the circuit, which is necessary for deciding where to put a symbolic unknown when one is needed.

Most devices have parameters; for example, a resistor has a resistance and a transistor has a polarity. These parameter's values are recorded, if known, by facts like (= (RESISTANCE R1) 1000.0), which says that R_1's resistance is 1000 Ohms. They do not have to be specified, and EL can be "back-driven" to deduce them if enough voltages and currents are known. An example of this use of EL to aid the choice of device parameters is given in the appendix.

The connections of the circuit are described by assertions like (CONNECT N54 (#1 R1)), each naming a single node and a single device terminal. These assertions need never be entered by an EL user, however, because we supply a special input format for conveniently defining devices and wiring them up into circuits (see Appendix).

Each type of device has conventional names for its terminals. For example, a resistor's terminals are known as #1 and #2; a transistor's are called E, B and C. The conventional terminal names have to be used because they are the ones that the laws for the device know about. It would be easy to wire a resistor up by its #3 and #4 terminals, but the EL law embodying Ohm's Law would not know about them.

The knowledge EL accumulates during the analysis of a circuit involves mostly the values of the voltage at or the current through particular device terminals. They are represented by assertions such as (= (VOLTAGE (#1 R1)) 10.0). The values of symbolic unknowns, when learned, are stored in the form (VALUE X15 15.4).

Perhaps the simplest circuit rules are those, such as Ohm's law, which can be represented by algebraic equations. In ARS such a law can be written very simply:

```
(LAW DC-OHM ASAP ((R RESISTOR) V1 V2 I RES)
    ()
    ((= (VOLTAGE (#1 !?R)) !>V1) (= (VOLTAGE (#2 !?R)) !>V2)
     (= (CURRENT (#1 !?R)) !>I) (= (RESISTANCE !?R) !>RES))
    (EQUATION '(&- V1 V2) '(&* RES I) R))
```

This is the EL law that implements Ohm's law. Like all EL laws, it has an arbitrary name, a set of slots or antecedent patterns to control its invocation, and a body which in this case consists of an algebraic equation. The name, chosen by us for mnemonic significance, is DC-OHM. ASAP indicates its invocation priority, which is normal, as it is for all laws that are simply equations between circuit parameters. DC-OHM declares the local variables V1, V2, I and RES to hold the two terminal voltages, the current, and the resistance value of the resistor. In addition, the type-restricted local variable R is used for the resistor about which the deduction will be made. The long list beginning with (= (VOLTAGE ... contains the demon's <u>trigger slots</u>. Their purpose is dual: to provide patterns to direct the invocation or <u>triggering</u> of the demon, and to gather the information needed in applying Ohm's law once the demon is invoked.

The ARS antecedent reasoning mechanism will signal DC-OHM whenever a fact is asserted that matches any of DC-OHM's trigger slots. DC-OHM itself then automatically checks all of its trigger slots to see which ones are instantiated and which ones are not. That information is passed to the function EQUATION, whose job is to deduce whatever it can from the equation it is given. If one of the terms in the equation (I, V1, V2, and RES, in this case) is unknown, EQUATION can deduce it from the others. If all the terms are known, EQUATION checks that they actually satisfy the equation, and if any of them is an algebraic expression involving symbolic variables, EQUATION can solve for one of them. Whenever EQUATION asserts a conclusion, it automatically records the instantiations of the trigger slots as the antecedents of the conclusion.

Notice the () before the list of trigger slots in DC-OHM. That is the list of <u>mandatory slots,</u> of which in this case there are none. Mandatory slots are just like trigger slots except that the law is not processed unless *all* of them are instantiated. DC-OHM's slots are not mandatory, since if any single one is missing DC-OHM can accomplish something by deducing a value for it. Mandatory slots are useful when a law is contingent on some fact. For example, different laws apply to conducting transistors and cut-off transistors. EL represents the knowlege that a transistor is cut off with an assertion such as

```
(DETERMINED (MODE Q1) CUTOFF)
```

When a transistor is cut off, no current flows into any of its terminals. One law, DC-BJT-CUTOFF-IC, enforces the absence of collector current:

```
(LAW DC-BJT-CUTOFF-IC ASAP ((Q BJT) IC)
    ((DETERMINED (MODE !?Q) CUTOFF))
    ((= (CURRENT (C !?Q)) !>IC))
  (EQUATION 'IC 0.0 Q))
```

This law has a mandatory slot requiring that the transistor in question be cut off. If that is known, the law will be applied and will deduce that the collector current is zero. If that is not known, the law will never be applied. Note that the slot that detects a known value of the collector current is *not* a mandatory slot, and its only function is to make sure that such a known value will be noticed by the law and checked for consistency.

The Method of Assumed States

The propagation method can be extended to any devices with laws that are invertible: if one terminal voltage or current is in fact fixed when others are given, then an algebraic expression for it in terms of those others may be needed in the course of propagation. Moreover, the expression must be "tractable", in the sense that the (human or mechanical) algebraic manipulation system may need to substitute in it, simplify it, or even solve it for unknowns appearing in it, in order to carry out the solution. For example, handling a diode is too complicated, since it would create the need to solve exponential equations. But even an "ideal diode" - a piecewise-linear approximation to a real diode - is too complicated to be handled symbolically as fluently as is necessary. It would introduce conditionals and "max" and "min" functions into the expressions, and they are not invertible.

But if the algebraic manipulation technology can't handle the device's laws as a whole, stronger methods of reasoning can break them down. Electrical engineering has a method known as the "method of assumed states", which is applicable to piecewise-linear devices such as ideal diodes. It involves making an assumption about which linear region the device is operating in (for a diode, whether it is "on" or "off"). This makes the conditionals simplify away, leaving tractable algebraic expressions to which propagation of constraints applies. Afterwards, it is necessary to check that the assumed states are consistent with the voltages and currents that have been determined.

For an example of such reasoning, consider the diode and resistor in series: Assuming
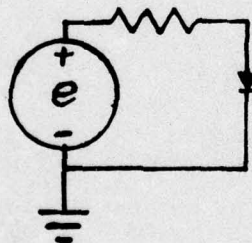
the diode to be nonconducting, we would deduce that there is zero current flowing, and that the voltage at the midpoint equals e. Since e is positive, that contradicts the conditions necessary for the diode to be off, as we assumed. On the other hand, if we assume that the diode is conducting, we deduce that the voltage at the midpoint is zero, and can then determine the amount of current. The current is flowing downward through the resistor and diode, which is consistent with the assumption that the diode is conducting.

When this method is mechanized, it is necessary to cycle through all of the possible states (linear regions) of the device, testing each one for consistency with the voltages and currents that follow from it. When there are several complicated devices, it is necessary to consider all combinations of all different states for each device. This causes an exponential explosion of the number of states of the system that must be investigated.

For example, this circuit
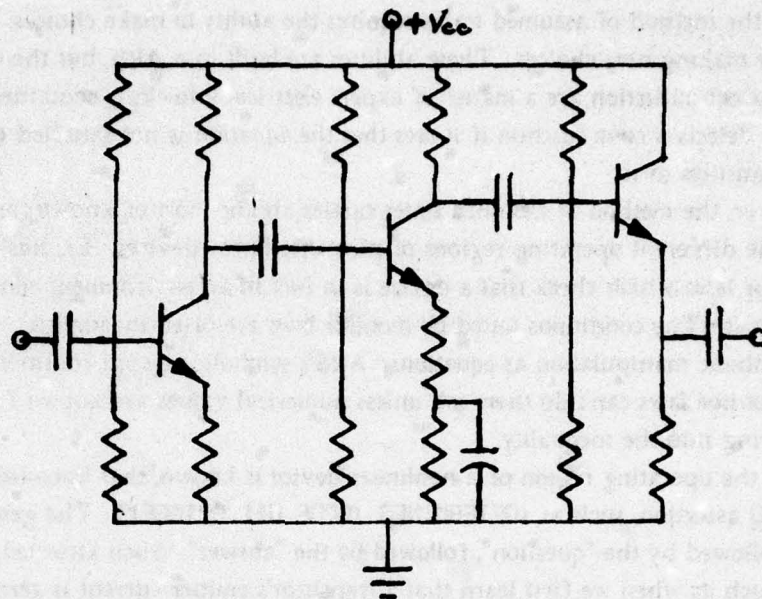
has three transistors. If the transistor model admits three states, active, cutoff and saturated, then there are, at first glance, 3⋅3⋅3 or 27 different triples of states that must be considered, of which only one (all three transistors active) is self-consistent. But actually, the states of the transistors are completely independent; the stages are coupled only for AC signals and have no effect on each other's bias conditions. Knowing this, we can find the correct state for each transistor separately, giving only 3+3+3 or 9 assumptions to be tested. Such situations are very frequent, and their detection is an effective way of reducing the work entailed by the combinatorial search for the correct states.

Making Choices

Using the method of assumed states requires the ability to make choices, and to handle contradictions by making new choices. These abilities are built into ARS, but the conditions for the detection of a contradiction are a matter of expert electrical knowlege, contained in EL laws. An equation law detects a contradiction if it sees that the equation is not satisfied by the known values of the quantities in it.

However, the method of assumed states carries another sort of knowlege, of the boundaries of the different operating regions of piecewise-linear devices. EL has special laws known as <u>monitor</u> laws which check that a device is in fact in an environment consistent with the state assumed for it. The conditions tested by monitor laws are often inequalities, which are not as conducive to symbolic manipulation as equations. ARS's symbolic algebra routines are helpless with them, so monitor laws can't do their job unless numerical values are known for all the parameters entering into the inequality.

When the operating region of a nonlinear device is known, that knowledge is represented by a <u>DETERMINED</u> assertion, such as (DETERMINED (MODE Q1) CUTOFF). The general form is DETERMINED, followed by the "question", followed by the "answer". Such knowledge might have been deduced (such as when we first learn that a transistor's emitter current is zero, and then deduce that it must be cut off), or it might be the result of an arbitrary choice. In the latter case, the choice *itself* is represented by a similar <u>CHOICE</u> assertion: (CHOICE (MODE Q1) CUTOFF), from which we pretend that the DETERMINED assertion was "deduced". The reason for having the two different assertions is that all the transistor laws that depend on the transistor's state can look for the DETERMINED, and thus work no matter how the known state was arrived at, while the backtracking mechanism can look for the CHOICE, and avoid trying to choose other answers for a question whose answer is not in doubt.

In fact, for the sake of efficiency, we have lumped together the state conditions not by state but by the circuit variable they test. Here is the monitor that checks transistors' collector currents for consistency with whatever state is assumed.

```
(MONITOR-LAW IC-MONITOR-BJT HIPRI-ASAP ((Q BJT) (IC NUMBER) (BE-DROP NUMBER))
     ((= (CURRENT (C !?Q)) !>IC) (= (BE-DROP !?Q) !>BE-DROP))
     ()
     (COND ((APPROX IC 0.0)
             (OBSERVE "(DETERMINED (MODE ,Q) CUTOFF) ANTECEDENTS))
           ((< (&* IC BE-DROP) 0.0) (CONTRADICTION DEMON ANTECEDENTS Q))
           (T (ASSERT-NOGOOD "(MODE ,Q) 'CUTOFF ANTECEDENTS NIL))))
```

It is called a MONITOR-LAW instead of just a LAW so that it will insist on having numerical values for the local variables declared to need them, IC (the collector current) and BE-DROP (whose sign indicates the polarity of the transistor). A zero collector current is consistent with only one state, CUTOFF. The function OBSERVE reports a contradiction to ARS if the transistor is in any other state. In addition, as a timesaving measure, if the transistor's state has not at the moment been chosen, OBSERVE chooses CUTOFF since it is the only consistent choice. If IC and BE-DROP have

opposite signs, the collector current is flowing backwards through the transistor, which is impossible in any state; in that case, a contradiction is reported to ARS for processing. Otherwise, there is a physically possible, nonzero collector current, which is consistent with any state *except* CUTOFF. The function ASSERT-NOGOOD reports that to ARS, causing a contradiction if an assumption of CUTOFF is currently in force. If not, a NOGOOD assertion is created (see **Contradictions**, below) so that future search through the space of state-combinations will be limited.
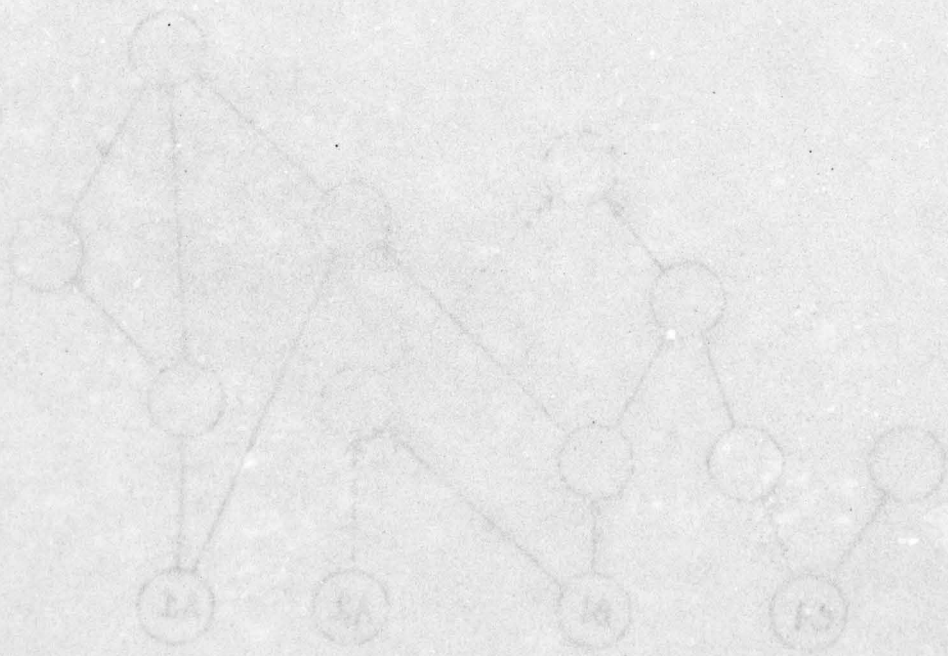
Dependencies and Contexts

The method of assumed states requires that ARS be able to reason from hypothetical assumptions. In addition, intelligent processing of contradictions involves distinguishing the guilty assumptions from the innocent ones. ARS's dependency records play a central role in both activities.

ARS keeps complete records of every deduction that it makes. The premises of the deduction can be found from the conclusion, and the conclusion from the premises. These records are used by ARS for several purposes: explaining the derivation of a fact to the user, finding the choices relevant to a contradiction, and delineating those facts which are currently believed to be true. A fact is believed (_in_) if it has well-founded support from atomic assumptions which are currently believed. An assumption, such as an arbitrary choice of a device operating region, may become disbelieved, perhaps because of a contradiction involving it. A fact which does not have well-founded support from believed assumptions is said to be _out_. If a choice (and its consequences) which has been _outed_ returns to favor, we use the dependency information to save the effort of reinventing implications from scratch. This process is called _unouting_. At any time, those facts actually believed are said to be _in_, while those under a cloud are _out_. Dependency information remains forever, even as the facts involved rise or fall in favor.

Here is a picture of the contents of an ARS fact data base, containing several atomic facts (device-state choices, or circuit construction specifications) and sundry consequences of them, showing a particular context selected. Al, Bl and Cl are atomic data that are currently _in_. Suppose Al and A2 are device-state assumptions, and in fact are alternative assumptions about the same device, so when Al is _in_, A2 must be _out_.
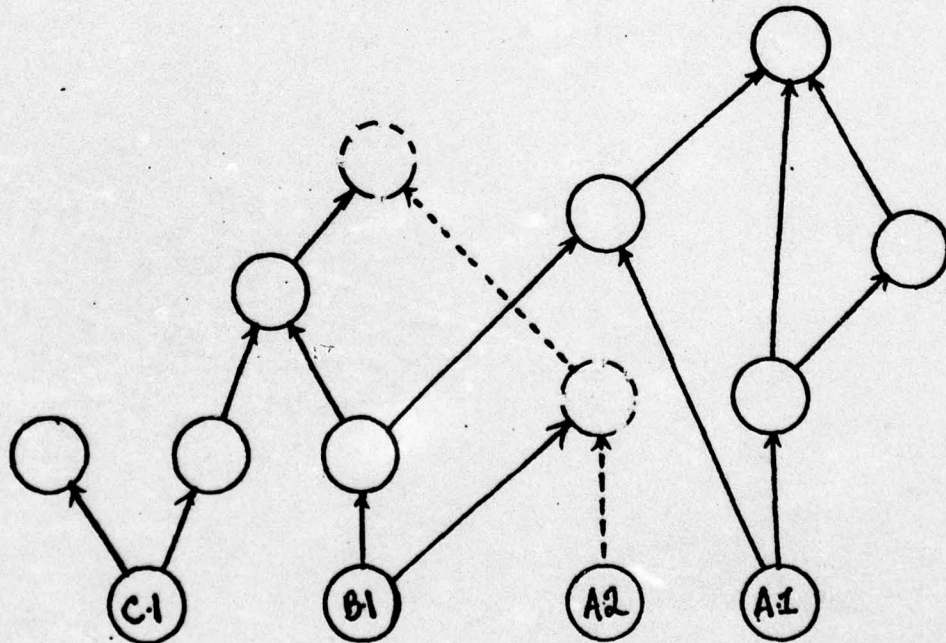
If A1 were to be retracted, the fact garbage collector would be invoked, leaving the data base as follows:
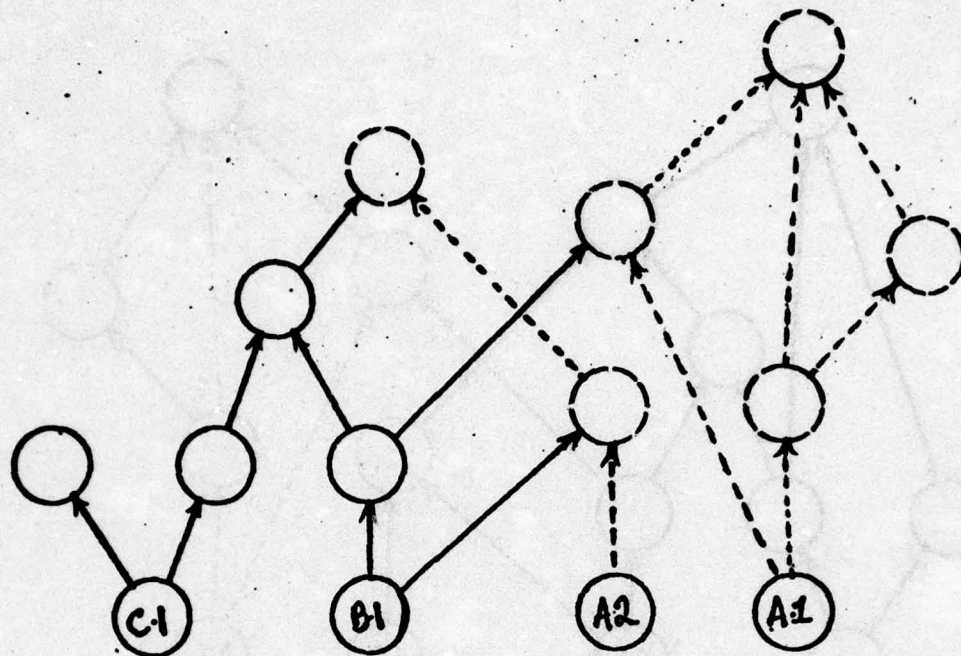
Believing A2 instead of A1 would cause unouting of several consequences:
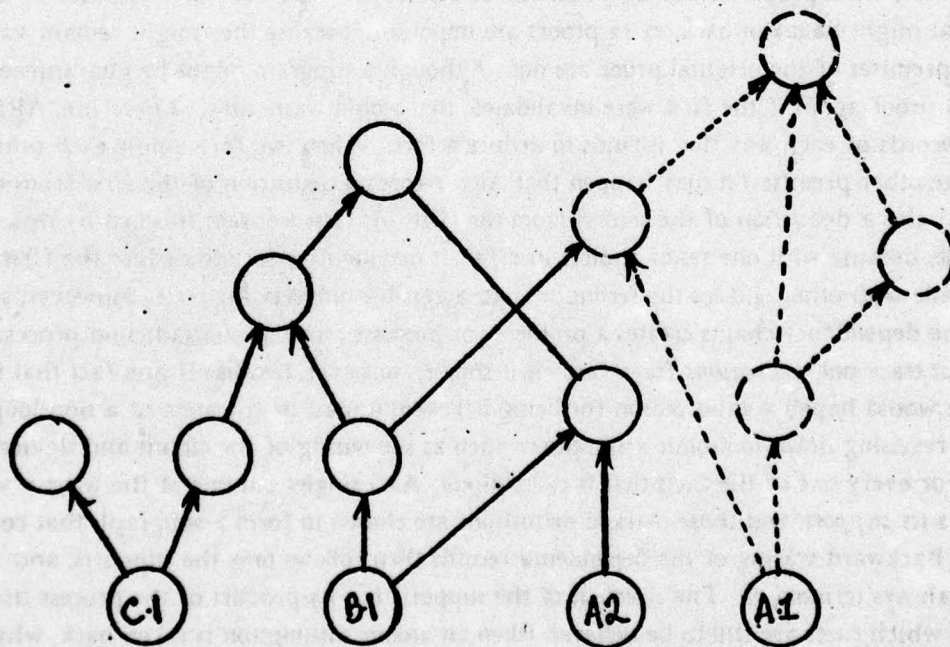
In addition, deduction would have the chance to add more facts:
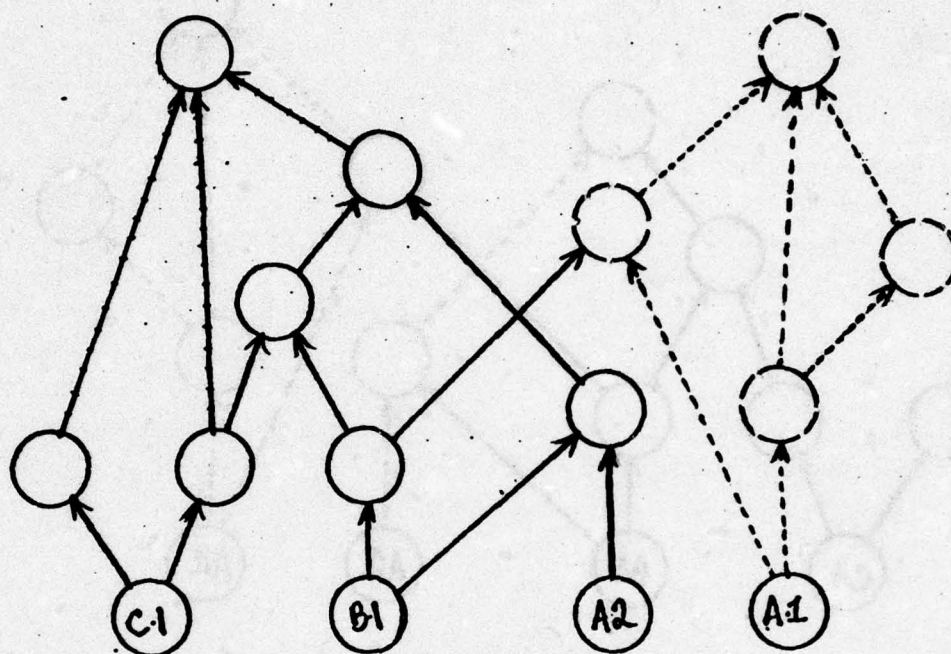
It may happen that a fact can be deduced in more than one way. If one is certain of one's premises, extra proofs of a known fact can be discarded. When the premises can be arbitrary choices that might be taken back, extra proofs are important, because they might remain valid when the premises of the original proof are not. Although a program might be guaranteed to find the second proof again if the first were invalidated, that would waste time. Therefore, ARS keeps separate records of each way that it finds to deduce a fact. When two facts imply each other (with appropriate other premises) it may happen that ARS records a deduction of the first from the second and also a deduction of the second from the first. At first we were shocked by this, but it is inescapable, because with one set of choices in effect it may be necessary to deduce the first via the second, while with other choices the second may be accessible only via the first. However, such loops in the dependency chains creates a problem for processes such as contradiction processing which must trace out the *reasons* for a fact. It is soluble, however, because if any fact that is *in* has (as we would hope!) a valid reason for being believed, it must be the apex of a non-looping subgraph reaching down to atomic assumptions such as the wiring of the circuit and device-state choices. For every one of the facts that is currently *in*, ARS singles out one of the ways it was deduced as its *support*, and those marked deductions are chosen to form a subgraph that contains no loops. Backward tracing of the dependency records then follows only the supports, and therefore always terminates. The selection of the supports is a by-product of the process used to determine which facts are still to be believed when an atomic assumption is taken back, which operates by scanning forward from the atomic assumptions currently believed and marking all their consequences as a garbage collector would. The facts that are marked become *in*, and the garbage facts are *out*.

Because facts and dependency information are never totally forgotten even when they are disbelieved, a short cut is possible when a once believed but later invalidated fact is validated once more. In a process known as unouting all the old consequences of the vindicated fact are re-examined, and if their other antecedents are all currently believed they too are marked in. If not for unouting, those consequences would be rededuced eventually anyway by the laws that originally deduced them, but unouting is much faster.

Taken together, the two processes of outing and unouting can be viewed as context-switching between contexts each associated with some subset of the set of possible assumptions. Since there are so many contexts to visit, the facts are not marked with the names of the contexts they are in; instead, context switching must identify and mark the facts that are in the context being entered. These contexts form a tangled hierarchy instead of the usual tree. Entering a subcontext is believing an additional assumption, which is done by unouting together with normal deduction. Moving to a higher context is disbelieving some assumption; this is done, by means of the fact garbage-collector, when contradictions happen. Note that, as in Conniver, a "subcontext" generally contains more assertions, and a "supercontext" contains fewer.

In fact, one could imagine the entire set of contexts as existing initially, but with no knowledge in any of them save the initial assumptions; then reasoning happens, adding more knowledge which is automatically placed in the context(s) determined by the premises of each deduction. Contexts in which contradictions are believed are the ones ruled out as acceptable solutions to the problem. The others are fertile ground for extension of knowledge. This point of view raises one above the petty details of when one should backtrack (try working a different context), and also raises the possibility of switching contexts simply because they appear unfruitful, and not necessarily contradictory (but this has not been implemented). Also, since backtracking is just a change of the point of attack, it loses no information.

A fact does not itself belong to any context, although it may be known in a particular context, or in several contexts for independent reasons. The dependency records also do not belong specifically to the context that was in effect when the deduction was made. Because of that, they are always ready to pull the consequences into a new context if the antecedents are deduced in it. Although a fact deduced is certainly in the currently selected context, it may also automatically, by virtue of its dependency links, be in some supercontext of it (and also many sibling contexts), if its proof did not make essential use of all of the atomic facts that define the currently selected context. In other words, a fact when deduced is not simply "installed" in the currently selected context, the way it would be in Conniver; it goes automatically into the highest supercontexts that its derivation will work in, and is merely "inherited" by the selected context.

This context mechanism might be applied to the understanding of processes that develop over time by associating a unique atomic datum with each significant local event occurring in the process. There would then be, for each stage of the process, a context in which the state of the system at that stage was known. However, the understanding of the process would not be tied to any global "time coordinate", and would not require any spurious time-ordering between events that were not causally related. Any "spacelike hypersurface" slicing through the process would have a corresponding context; time would be a decentralized object that could be advanced locally in one region of the process while being "left at the same instant" in other regions.

Contradictions

When EL uses the method of assumed states to analyze circuits containing nonlinear devices, incorrect assumptions are detected by means of a contradiction, which is the specific event in which the chosen assumptions are seen be inconsistent. A contradiction is detected by a particular law - most often by monitor laws that exist for just that purpose. Contradictions are remembered both by contradiction assertions which are placed in the dependency-structure at the point of contradiction, and by NOGOOD assertions which record essentially the same information in a form easily used by the routines which choose alternate state-assumptions. A contradiction assertion does not explicitly contain any information; its significance lies entirely in its list of antecedents. A NOGOOD assertion explicitly lists the state assumptions that conspired to produce the contradiction. A typical contradiction might depend on dozens of atomic facts, including some device-state choices such as (CHOICE (MODE Q3) BETA-INFINITE) and (CHOICE (MODE D2) OFF), as well as many circuit construction details such as (RESISTANCE R1 1000.0) and (CONNECT N54 (B Q1)). The contradiction assertion would have all of them as antecedents (indirectly); the NOGOOD assertion might be (NOGOOD ((MODE Q3) BETA-INFINITE) ((MODE D2) OFF)), and its antecedents would include the RESISTANCE and CONNECT assertions but not the CHOICEs.

When we view sets of assumptions as determining contexts, a contradiction is a fact deduced in a specific context, which shows that context (and all of its subcontexts) to be of no further interest. But if the fact of the existence of the contradiction is to be available for use (such as in controlling a search), that fact must reside in a different context. NOGOOD assertions fill that role. The simplest way to remember the contradiction's existence would be to assert a fact containing a list of all of the atomic assumptions of the contradicted context, found by walking back through the dependency tree from the contradictory facts (or, just as good, from the contradiction assertion). In the example we are using, the RESISTANCE, the CONNECT, and both CHOICEs would be listed in the NOGOOD, which would have no antecedents. Since such a NOGOOD would be true regardless of the truth of any of the premises it listed, it would exist in the highest context of all, which is the one that depends upon no atomic facts. However, (not (A and B)) can also be stated as (A implies (not B)). Any subset of the basis of the contradiction can be de-emphasized by being made antecedents of the NOGOOD rather than part of its list. In our original example, the RESISTANCE and CONNECT assertions were de-emphasized. De-emphasis makes the information totally unavailable in some contexts (those that do not include the de-emphasized antecedents), but by the same token reduces the number of NOGOODs that are in at any moment, and also reduces the size of each NOGOOD's list. That is valuable, since whenever ARS needs to choose a state for a device it must examine all NOGOODs that are in, to eliminate choices already known to be incorrect; each NOGOOD must be processed to see whether it lists the choice under consideration, and whether the other atomic facts it lists are currently also in. De-emphasizing some of the atomic facts causes the normal context mechanism to help with this filtering of the NOGOOD assertions.

Of course, de-emphasis can have drawbacks. The most extreme possible de-emphasis would leave only one assertion in the NOGOOD's list, while all the others became antecedents of the NOGOOD. This would make the NOGOOD almost useless for pointing out contexts which were not worth visiting. Imagine that AI, BI and CI are atomic facts that lead to a contradiction, and that

Al is listed as no good, with Bl and Cl as conditions. If later Al and Bl were *in*, and Cl were under consideration for belief, that NOGOOD assertion would be *out*, and there would be no understanding that Cl led to a context already tried and discarded. The NOGOOD would not be performing its intended function. This would not be a disaster, since bringing Cl *in* would bring back the original contradiction assertion also by un*out*ing, but much time might be wasted. This "thrashing" is most painful with such excessive de-emphasis, but any de-emphasis has the ability to cause thrashing if its implicit assumptions about the relative stability of the atomic facts prove to be wrong.

A practical system must find a compromise between thrashing, and examining too many too-big NOGOODs whenever a choice is made. We do not know of any domain-independent solution to the problem. What is clear is that it is good to de-emphasize facts that are unlikely to change. In the domain of electronic analysis, we emphasize device-state choices, and de-emphasize circuit wiring and intrinsic device parameters such as the resistance values of resistors, since during the analysis of a specific circuit the latter usually do not change. In circuit design there might be occasions when some circuit voltages that represent the design criteria would be least likely to change, device-state assumptions would be guessed at next (along with approximate circuit wiring), and placement and values of resistors and capacitors would be chosen last. Then, NOGOOD assertions might emphasize the detailed wiring decisions and de-emphasize the large-scale ones. A second level of NOGOOD assertions might speak of overall circuit plans, de-emphasizing only the design criteria. Thus, the level of a contradicted context in the tangled hierarchy would guide the choice of a context for the NOGOOD.

Compound Devices, and Identified Terminals

Engineers often think of a subcircuit as a "black box". A truly black box -- one whose insides are hidden, such as an op-amp -- is intellectually (and computationally) just an element. More interesting is a "grey box" which may be ambivalently thought of as a black box or as a configuration of components. Grey boxes are often used to summarize some aspect of the behavior of a configuration as a whole. It is economical to store the most important features of the behavior of common configurations as grey-box laws so that they do not have to be computed from scratch each time the configuration is used. Sometimes, in fact, there are laws about the behavior of a configuration which are crucial to analysis of circuits containing it, but which are very difficult to derive from the behaviors of its components.

For example, some common configurations of transistors cannot be understood in terms of EL's simple-minded model of transistors. Such an application is the emitter-coupled-pair (ECP):
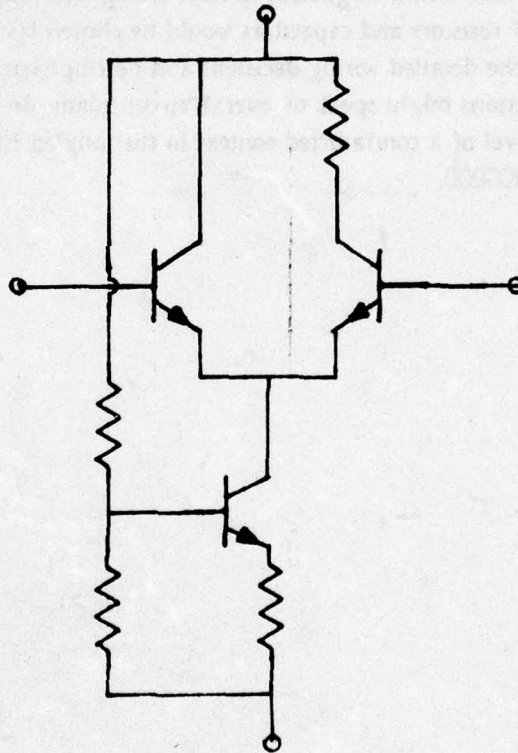
The problem here is that one must use the exponential diode model of the transistors to derive the fact that (in a correctly biased ECP) the incremental difference in collector currents is proportional to the difference in incremental base voltages (and furthermore, that constant of proportionality is almost independent of the transistor characteristics).[Derivation] We could try to solve this problem by attempting to include the exponential diode model of the transistor and the algebraic expertise required to use it. Notice, however, that this important, but difficult to derive fact about the currents in an ECP is in itself a simple linear law for the ECP. Grey-box laws allow us "package" this fact about the configuration so that it can be used in analysis, without increasing the

complexity of our basic transistor model. In this way we can use the much simpler (but less accurate) models in most cases and have a way to impose the constraints on the configurations that depend on the more accurate models.

Another problem is that there is not just one ECP circuit. For example, here is another variety of ECP:
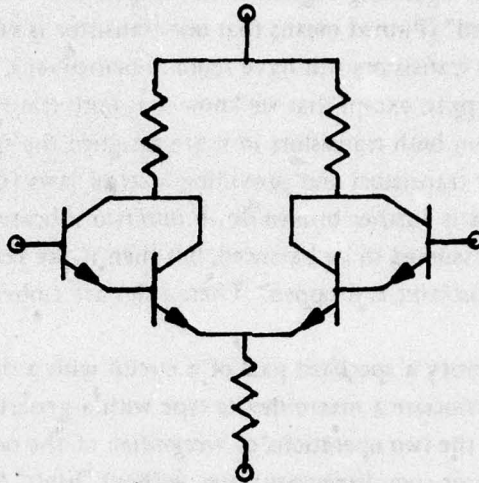
If we had to specify a distinct set of grey-box laws to cover each instance of ECP we would be fighting a losing battle. We must express the laws to capture the essence of ECP and not any particular circuit.

The notion of a grey box is implemented by the EL macro-device feature. An EL user may specify that a particular subcircuit is to be viewed as a grey box. EL has the ability to identify a portion of a circuit with a macro-device on a terminal-by-terminal basis. The subcircuit and the macro-device are regarded as alternate descriptions of what is connected to their common environment. The terminals of the macro-device are each identified with a terminal of one of the devices in the subcircuit. The device is allowed to have unidentified connections, and so is the subcircuit - in fact, to speak of the subcircuit is slightly misleading, since it suggests that there must be a boundary surface that divides the circuit into "inside" and "outside", and no such is required. When two terminals are identified, EL assumes that they must have the same voltage and *the same current*. This is not the same as connecting terminals at a node!

The macro-device is then just a method of attaching extra laws which supply additional constraints among the voltages and currents on the identified terminals and thus help determine the circuit unknowns. Since whatever is learned about a terminal of the macro-device is automatically propagated to the identified terminal of the subcircuit, the two sets of laws can stimulate each other. If they disagree on their conclusions, a contradiction occurs.

When a macro-device, such as ECP, models some aspect of a configuration of devices with nonlinear properties, the special laws of the macro-device may be contingent on the operating regions of its components. It is also often true that the whole configuration has fewer consistent states than one would calculate from the those of the parts taken independently. Furthermore, as

In the case of the ECP, there may not be any assignment of regions (using the simple transistor model) to the components which is consistent with the known behavior of the configuration.

This set of problems is resolved by allowing a macro-device to have a set of operating regions of its own, on which its own laws are contingent. Furthermore, the macro-device must be able to control the assignment of operating regions to its components. Thus, for example, an ECP may be either "active" or "pinned" (Pinned means that one transistor is either saturated or cutoff whereas active means that both transistors still have room to maneuver.). If the ECP is pinned, the normal laws of transistors apply, except that we know that both transistors cannot both be active. If the ECP is active, then both transistors in it are assigned the special region "ecp-active", suspending the normal laws for transistors and providing a set of laws for this special situation. In the ECP case, the active state is further broken down into two subcases -- it is either "balanced" or "unbalanced". The ECP is assumed to be balanced, but then if the rest of the circuit unbalances it, the balancing constraint is dropped. These rules are embedded as special laws for the ECP.

Although EL can identify a specified part of a circuit with a device of a specified type, it does not have the ability to associate a macro-device type with a generic equivalent circuit. Such an ability would make possible the two operations of *recognition* of the occurrence of such macro devices as emitter-coupled-pairs or complementary pairs, without "hints" from the user, and *expansion* of a macro-device into an identified equivalent circuit. The latter operation might facilitate the use of an EL-like subprogram by a circuit-design program (either entirely automatic, or merely assisting a human). In addition, it might be applied to devices such as transistors, which are not "really" macro-devices, but which an equivalent circuit might at times allow EL to model more accurately.

However, such operations are more complicated than they seem. An equivalent circuit recognizer that would accept only a specific exact pattern of circuitry would be of very little use. This is because the "circuit" of such an application as the ECP is so unconstrained; there can be many different arrangement of components in between the two transistors, that will have slight but useful effects on the behavior of the circuit without making it useless to view the configuration as an ECP. Thus, the recognizer must be able to be flexible in its criteria. The expansion operation is even more problematical, for while the recognizer can accept many circuit configurations, the expander must choose one appropriately, or give the user the ability to specify changes in almost all the details of its expansion. But making the user specify too many details would deprive the expansion operation of its usefulness.

The Queue-based Control Structure

The gross-scale control structure used in ARS is event-triggered, as in a production system or a Markov algorithm, rather than sequential as in a classical programming language. Sequential control structure is confined to the inside of a law or demon; demons cannot transfer control to other demons, but only return to the scheduler, which has queues of demons to run and arguments to feed them. Demons can affect the future actions of the program only by adding to the queues, but even that is not done directly. Instead, demons assert, and the process of assertion enqueues demons whose trigger-patterns match the newly asserted fact. This produces a degree of isolation for each individual demon, automatically making most device-law demons very modular. It also obviates a great deal of decision-making that would otherwise have to go into the sequential algorithms for handling many local circuit configurations. Circuit analysis must deal with many different types of building blocks, strung together in any order. Without the queue, the overall structure of the deduction process would be that of a loop containing a single many-way dispatch that decided what type of deduction was appropriate to perform next.

There are actually several queues for demon-invocations, with different priorities. Each demon specifies which queue it should go on. There are three queues used for DC analysis. Most demons, including equation-laws, are intended primarily for deducing new facts. They go on the middle priority or normal queue. Monitor demons, which exist mainly for finding contradictions, go on the high priority queue. That is because if there is no contradiction then all the demons will be executed eventually anyway, so their order makes no difference; if there is a contradiction, then the faster it is found, the less time it wastes. The low priority queue is used for choosing device-state assumptions, because it is best to explore all the consequences of one assumption before making more assumptions, in case there is a contradiction. Moreover, the possible states for a device can sometimes be narrowed down by knowledge about the device's environment. Given the opportunity to try deduction or to assume a device's state, it is better to make the deduction first, since they must both be done eventually, and the deduction has a chance of reducing the amount of work involved in finding the correct device-state if the deduction is done first.

There are three more queues for AC analysis, with lower priority than the DC queues. That is because the EL laws make it is very unlikely for a contradiction to involve any of the AC analysis; if the DC analysis finds none, there probably is none. Again, it pays to avoid doing any AC analysis for states that are going to be ruled out anyway.

Unfortunately, the queues of ARS are a very sensitive data structure. If any demon that in fact ought to be run is missing from the queue, nothing will ever detect that fact, or put it back on the queue, since that could be done only by the assertion of the fact that can trigger the demon, and said fact is already asserted. Such problems are not hard to avoid when only straightforward propagation is involved. The real difficulties come with contradictions. They are of two kinds: those accompanying the forgetting or *outing* of facts, and those that pertain to the very demons which detect the contradictions.

When a demon in ARS detects a contradiction, it drops what it was doing and makes a contradiction-assertion instead, causing the contradiction to be processed immediately. In some cases, that is guaranteed to cause the *outing* of at least one of the facts on which the demon's operation depended. Such cases are non-problematical, if (as is usually the case) the demon will be

incapable of doing any useful work until a similar fact is later asserted, and such an assertion will enqueue the demon in the normal manner. However, not all demons have that useful property. A case in point is that of the monitor demon IE-MONITOR-BJT, which examines the emitter currents of all transistors. If the emitter current is zero, and the transistor is currently assumed to be active, the monitor detects a contradiction. If there is no assumption in force at the moment about the transistor's state, the demon asserts that the transistor is cut off. Thus, if the demon is run because the emitter current has just been asserted to be zero, and a contradiction is detected, the demon really ought to be run again so it can make "cut off" the new state. To make that happen, the demon explicitly requeues itself.

Most laws are simply equations relating circuit parameters (Ohm's law is an example). The normal case in which such a law can do useful work is when all but one of the parameters it connects are known. In that case the unknown one is determined from the others. If more than two are not known, the demon is helpless. Because of that, no special action is necessary if one of the known parameter values is *outed*. But it can also happen that all of the parameters are known, but at least one of them is an algebraic expression containing a symbolic unknown. The equation of the law can then be used to solve for the value of the unknown. This is fine and dandy until one of the circuit parameters' values is *outed* because of backtracking. After that, though there is not enough information any more to solve for the symbolic unknown, it is still possible to compute the missing parameter from the others. For this, the demon must be run again. That is brought about using the mechanism of forget-functions. Any assertion can have a forget-function, which will be called whenever the assertion is forgotten or merely *outed*. In this example, a special assertion called the CHECKED assertion is placed in the dependency-chain between the assertion of the value of the symbolic unknown and the facts used as the demon's antecedents. Its only use is to hold on to a forget-function that will requeue the equation-demon if the CHECKED assertion ever vanishes (because one of the circuit parameters in the equation was forgotten). In fact, the CHECKED assertion is necessary even if there is no symbolic unknown in the circuit parameters' values, since it is still the case that if one parameter is forgotten it can be rededuced from the others. Note that in this case the equation of the demon has given no new information about the circuit, showing that the equation was algebraically dependent on the other equations describing the circuit. Such an event happens at least once per circuit, since at each node KCL, Kirchoff's current law, states that the sum of the incoming currents is zero, and that set of equations is not independent: KCL on any one node follows from KCL on all the other nodes.

Some monitor demons have the ability to predetermine the state of a device, eliminating the need for searching. For example, if I-MONITOR-DIODE sees a nonzero current in a diode, it can assert that the diode is in the ON state. When a device-state choice is *outed*, all the monitors that might be able to predetermine the state choice should be given a chance to do so. This is also implemented by a forget-function.

Monitor demons often check the signs of currents, or otherwise test inequalities. While equations are quite happy with algebraic expressions, inequalities are stymied by them (unless one uses a more sophisticated algebraic manipulation package than ours). For example, if a transistor's emitter current is assigned a value which involves a symbolic unknown, IE-MONITOR-BJT will be run, but will be unable to perform its function. Presumably that unknown's value will eventually be learned, and it is essential that IE-MONITOR-BJT be run again then, or else a contradiction

might go unnoticed and a false analysis be accepted. That is brought about by means of the HANGING assertion, which records the name of a demon (and its arguments) and the name of a symbolic unknown whose value the demon is waiting for. IE-MONITOR-BJT itself makes a HANGING assertion when it sees such an obstacle. Whenever the value of a symbolic unknown is determined, a check is made for HANGING assertions listing it, and the demons they mention are requeued.

The above examples are not the only cases in which demons need to be requeued; some are quite obscure. The examples described should suffice to show why the queue is a constant source of bugs. Because of that, we have given thought to possible mechanisms that might have the same modularizing ability as the queue, without its fragility. One which was actually implemented at one time was to keep the queue information in the data base of facts. The dispatcher which now removes entries from the queue and processes them, in that implementation used to scan the data base for possible places to run demons and run one. The only new information that had to be stored was the list of demon-invocations already performed -- in the form of a HAS-RUN assertion for each successful demon-invocation -- so that the dequeuer could avoid running demons redundantly. This eliminated all danger of accidentally losing an entry from the queue, since the whole queue was effectively recomputed each time it was used. It also made CHECKED assertions unnecessary, since the *outing* of a HAS-RUN assertion would automatically subject the demon to being rerun. Unfortunately, in our implementation, this mechanism proved to be unacceptably slow.^{Queueless}

The Data Base of Facts and Demons

ARS stores all problem-specific knowledge in the form of <u>facts</u> or <u>assertions</u> in an indexed data base. An example of a fact is (= (VOLTAGE (E Q1)) 1.3), which EL takes to mean that Q1's emitter voltage is 1.3 volts. (ALTERNATIVES (MODE DIODE) (ON OFF)) says that the device-state of a diode, known in EL as its MODE, has two possible values, called ON and OFF. The first sample fact is typical of many of the facts EL generates as it runs. The second is actually part of EL, and represents knowledge appropriate to choosing diodes' states.

Besides its <u>statement</u> (which is what "(= (VOLTAGE (E Q1)) 1.3)" is), a fact also has a unique <u>factname,</u> which is a LISP atom. The factname's LISP property list is used to record the fact's auxiliary information, such as its dependency records, whether it is currently believed, its forget-function (see below) -- everything other than just "what the fact says". In addition, the fact is referred to whenever possible by its factname (in dependency records, for example). It is tempting, when using a relational data base, to break all knowledge into small pieces and make each piece a separate assertion. That can lead to great inefficiency, as we discovered when using early versions of ARS. Since then, we have rewritten ARS to use the indexed data base mainly as a way of placing property lists on arbitrary LISP lists as if they were atoms. One might suggest that a simple hash table might serve, but that is in fact how the data base is implemented anyway.^{Equality}

EL records the names of the devices in a circuit with <u>IS-A</u> assertions, such as (IS-A R1 RESISTOR) or (IS-A NS4 NODE). An EL demon driven by those assertions controls an ARS mechanism for typed variables in the trigger slots of laws. Whenever an IS-A is asserted, a LISP property is placed on the device's name that identifies it as a certain type of device. The pattern matching mechanism that triggers demons then insists that a typed pattern variable (such as R, in the demon DC-OHM) match only the name of a device of the appropriate type.

<u>Demons</u> in ARS are programs subject to pattern-directed invocation. Each EL demon generally implements a single item of knowledge about electronics (though a few embody more general problem-solving knowledge). Here, for example, is the demon that embodies the fact that all of the current into one of a resistor's terminals comes out the other one. This law is needed because the fact data base is not constructed so as to retrieve (CURRENT (#1 R1)) and (CURRENT (#2 R1)) from the same place automatically.

```
(LAW DC-2T-R ASAP ((R RESISTOR) I1 I2)
    ()
    ((= (CURRENT (#1 !?R)) !>I1) (= (CURRENT (#2 !?R)) !>I2))
    (EQUATION '(&+ I1 I2) 0.0 R))
```

Its name, chosen by us for mnemonic significance, is DC-2T-R. ASAP indicates its invocation priority. DC-2T-R uses the local variables I1 and I2 to hold the two terminal currents. The long list beginning with (= (CURRENT ... contains the demon's <u>trigger slots</u>. Their purpose is dual: to provide patterns to direct the invocation or <u>triggering</u> of the demon, and to gather the information needed in applying the law once the demon is invoked.

When the function LAW, a LISP macro, is called to create the demon DC-2T-R, it stores information about the trigger slots in the demon data base, which has the form of a stylized decision tree which, applied to a fact, quickly finds those demons which have at least one trigger slot that matches the fact. Each of those demons has *one* of the facts it needs to be able to do useful work; it might or might not have all it needs. ARS enqueues them all for invocation, and the demons themselves must decide whether they can do anything. For that, they use the trigger slots again, applying them all as patterns to the fact data base. Thus, if (= (CURRENT (#1 R1)) 10.8) is asserted, DC-2T-R will be triggered, and the value matched by the part variable R will be remembered as an argument (the declaration of R as (R RESISTOR) will prevent triggering unless what R matches is actually the name of a resistor).

When the demon is invoked it will apply all of its trigger patterns to the data base, using its argument as the value of R during the match -- that is how we make sure that we find voltages, current and resistance for a single resistor instead of for four different resistors! Variables appearing in the pattern with the ">" operator have no effect on the triggering of the demon, but at the matching stage they are assigned whatever value they happen to match, *if* the pattern matches anything at all. Thus, if in addition to the triggering assertion about the voltage at (#1 R1), the two facts (= (VOLTAGE (#2 R2)) 0.0) and (= (RESISTANCE R1) 1000.0) were in the data base, DC-OHM's matching phase would set V1 to 10.8, V2 to 0.0, and RES to 1000.0. I would remain NIL if there were no assertion about the value of (CURRENT (#1 R1)).

In addition to setting local variables, the matching process places a list of the factnames of the facts matched in the variable ANTECEDENTS, along with the demon's demonname. If the demon asserts any new fact, it will normally supply that list as the antecedents of the fact. This is how the dependency records obtain the information of what other facts were used in deducing the new one.

After the matching phase, the body of the demon is executed. In this case the body is just a call to the function EQUATION, which does all the work of extracting any possible new information from the specified equation and the parameter values obtained by the match phase. It also knows how to report a contradiction if the parameters have values that can't fit the equation, and that there is nothing to be done if too few of the parameters are known yet.

How it works:

ARS has three different storage representations for the three main types of entities it knows: facts, demons, and dependencies. Dependencies are stored as simple LISP lists. Each fact's factname has a CONSEQUENCES property which is a list of the factnames of all the facts deduced from it, and an ANTECEDENT-LISTS property which is a list of lists, one list for each way the fact has been deduced, containing the factnames of the facts used in the deduction.

A demon is more complicated. In addition to the LISP function which implements its body it must enter its trigger slots in a data base that allows that slots that match a given fact to be found easily. ARS compiles the trigger slots into a decision-tree[Discrimination nets] which it builds incrementally. The tree specifies locations in the fact being matched against (such as, "the CAR of the CAR of the CDR"), and then various things to compare it against, each leading to some demons or to further decisions.

The fact data base is the most complex of the three. It is a bare-bones version of the

Conniver data base. It indexes each fact by each of the atoms in it, together with its position. Thus, the fact (= (VOLTAGE (B Q1)) 10.0) would be indexed under "= in the CAR", "VOLTAGE in the CAADR", "B in the CAADADR", "Q1 in the CADADADR", and "10.0 in the CADDR". This method of indexing makes it easy to look for all the facts that match a pattern which has some positions unspecified. In ARS's notation, !>FOO is a pattern which matches anything, and sets FOO to what was matched. (= (VOLTAGE (B !>Q)) !>V) ought to match any assertion about the voltage on the base of something. ARS can find all the facts that it matches by looking in the fact data base index under "= in the CAR", "VOLTAGE in the CAADR", and "B in the CAADADR", and intersecting those lists of facts.

Actually, the index-pairs of atom and location are hashed into a fixed length table, so a bucket may contain things that are irrelevant to the index-pair being fetched. For that reason, an actual matching test must be made on each fact that the indexer returns.

Originally, all three types of data were kept in the fact data base. Facts were kept as lists (<statement> FACT <factname>), demons as lists (<slot> DEMON <demonname>), and dependencies as lists (<fact1> DEPENDS <fact2>). This had the advantage of being easy to do, but was very inefficient. This was obviously so for dependencies, whose representation we changed shortly after ARS began to look at them during normal operation. It was not nearly so obvious that this was a bad way to store the trigger slots of demons. That was discovered only by timing measurements. The problem was due to the fact that the search operation for demon slots is different from the search operation for facts. Facts are searched for with a pattern like (= (VOLTAGE (B !>Q)) !>V), and wherever it has an atom, all the facts it matches must have the same atom. On the other hand, when a fact such as (= (VOLTAGE (B Q1)) 10.0) is asserted, the slots that should trigger might differ from the fact itself in any position. For example, (= (!>QTY !>TERMINAL) 10.0) should be triggered if it exists, as should (= !>ANYTHING !>VALUE), (!>RELATION (VOLTAGE (B !>Q)) !>NUMBER), (= (!>QTY (!>ECB !>Q)) !>VALUE), or even just !>FACT if any demon has it as a slot! The fact data base is poorly suited to that retrieval operation. The implementation of a separate demon data base resulted in a factor of two improvement in the speed of the entire system.

A lesson to be drawn from the experience with the ARS data base is that <u>it is usually worthwhile to factor the retrieval problem</u>. We had a data base whose entries could be divided easily into three classes, such that any retrieval would certainly be looking for entries in only one of the three. Now we have three data base systems, one for each class.

We think this corrected mistake is worth mentioning because it isn't a new one -- Conniver made the same mistake! IF-ADDED methods were stored in the fact data base and looked up just as ARS demons formerly were. People have often assumed that the slowness of Conniver was due to the basic interpreter, but in fact measurements showed that Conniver spent most of its time searching the data base. Our results with ARS suggest that Conniver also might run considerably faster with ARS's newer demon data base.

Conclusions

Our research strategy has been the application of artificial intelligence techniques to the construction of an expert problem solver in a non-trivial domain. We feel that this strategy has been very fruitful. We have developed two methods. One is a method of electrical network analysis we call <u>analysis by propagation of constraints</u>. The other is the technique of efficient combinatorial search by <u>dependency-directed backtracking</u>. Analysis by propagation of constraints would not have been developed in the absence of such artificial intelligence techniques as <u>symbolic manipulation of algebraic expressions</u> and <u>antecedent reasoning</u>. Dependency-directed backtracking is a new artificial intelligence technique whose development was stimulated by the needs of this exceptionally deep, yet well-structured domain.

Electrical circuits is an especially good domain in which to develop artificial intelligence techniques. Reasoning about circuits is deep enough to benefit by the application of powerful techniques, yet the problems are drastically simplified by the fact that the interactions between parts of a circuit are well-defined and constrained to occur by explicit connections. Another advantage is that it is clear whether or not an answer tendered is in fact correct.

An even more important reason for studying reasoning about electrical circuits is that such reasoning is typical of the reasoning done by all engineers, including computer programmers. We wish to understand the nature of reasoning about deliberately constructed systems. An understanding of the epistemology of engineering will enable us to make programs which can significantly aid the design and debugging of engineered systems, including computer programs. We expect that such an understanding will entail the development of techniques concerned with many aspects of reasoning, including constraints, causality, and teleology.

Generality and Extensibility

ARS is a language in which it is easy to embed problem solving rules for domains in which the solution to a problem may be obtained by the symbolic relaxation of local constraints. It provides special features for explanation of its conclusions to its users and it uses this ability to provide for reasoning hypothetically and for efficient combinatorial searches. We are convinced that the ARS paradigm is applicable to a wide variety of domains. Jon Doyle <Doyle 1976> has looked into the possibility of a set of ARS rules to prove theorems of plane geometry (no constructions) along the lines previously investigated by Nevins <Nevins 1974>. Matt Mason <Mason 1976> has constructed a set of rules for the analysis of the flow of materials and money in a hog farm.

EL, the set of rules of electrical circuit analysis, is a very pleasant system to work with. The explanations it provides to a user can be useful for helping understand the behavior of a circuit -- how particular device parameters affect the behavior of interesting circuit parameters. If a particular answer is surprising it is possible to find out why EL thinks that that answer is true. The user can have more confidence in these answers because he can check the reasoning. The complex programs of the future will most certainly have to use similar techniques so as to be responsible for their answers.

EL is very extensible because of the modularity imposed on it by the conventions of ARS. It is easy to add new device types to EL because the rules for the new elements can be

constructed in isolation and usually do not interact with the laws for the existing devices. This is even true in the case of macro-devices because the macro-device gets control of the selection of states for its parts. Gerald Roylance <Roylance 1975> created rules for enough macro-device types to allow the analysis of the 741 operational amplifier. In fact, it would be easy to add enough rules to create whole new analysis modes. To add sinusoidal steady state analysis, for example, would require only a set of laws which characterized parts in terms of impedances and a small extension to the algebraic manipulation package to allow the manipulation of complex quantities.

Problems and Plans for the Future:

There are many modes of reasoning about circuits which we have not captured in EL. Some of these, such as sinusoidal steady state analysis are really very simple extensions which we have just not yet gotten around to doing. Others, however are much harder and may represent more fundamental problems with the ARS paradigm for embedding of knowledge.

For example, it is not obvious how to represent the knowledge required to do time-domain analysis in ARS. In steady state, dc, or incremental analysis, it is possible to summarize the entire behavior of a network unknown as a simple, algebraically manipulable expression. Time domain analysis requires explicit time functions, some of which might be algebraically horrendous, to be manipulated. Gerald Roylance <Roylance 1976> is beginning to investigate what kinds of qualitative reasoning is required to bypass this algebraic roadblock.

Simple equation demons are an essentially declarative representation for equality constraints, even though demons are procedures. Inequalities are represented as monitors and cannot be manipulated easily. This leads to the following problem:
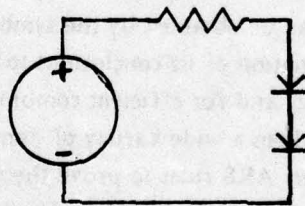
In this circuit, the only physically consistent states for the diodes to be in is both conducting. If, however, we assume both are cutoff, EL will not discover the contradiction because that would entail the propagation of inequalities. De Kleer <de Kleer 1976> solves this problem by propagating ranges, but he cannot handle symbolic expressions at all. We are looking for a way of handling inequality constraints in a more declarative manner than by monitor demons.

Another problem is that although EL can use grey-box laws which embody certain global abstractions, EL does not make use of the general equivalent circuit ideas in its analysis. Thus, for example, one can declare a particular circuit to be an amplifier by its identifying its input and output terminals with those of a special macro-device which specifies that the incremental voltage on the output is proportional to the incremental voltage on the input (perhaps with an unspecified gain). EL can then deduce the value of the gain by working out the incremental output voltage

for a given incremental input voltage. The problem is that this value of the gain <u>depends</u> on the particular incremental input voltage used to derive it. Thus it cannot be used to compute the incremental output for any other value of input since for that value of input the gain would be <u>out</u>. This is basically a problem of the interaction of contexts with logic. We need a mechanism by which the value of the gain can be made to depend on <u>the reasoning behind</u> the value of the incremental output voltage for the given input voltage rather than their values.

     A related problem is what we call "anomalous dependencies". Consider the following situation:
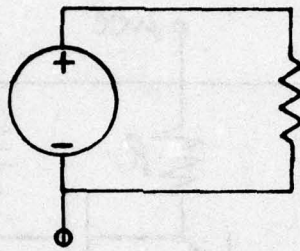
In this circuit the current through the resistor is independent of the value of the voltage at the **bottom** node of the circuit. The justification of the value of the current produced by the EL laws **will, however,** include this voltage because it was propagated through the voltage source to produce **a voltage** at the top. Ohm's law was then applied to the difference of the voltage at the top and **bottom** nodes. These extra dependencies can increase the amount of search the program must go **through** because they can introduce extra assumptions in the proof of a contradiction. We have **not, however,** had this cause much trouble in any circuit we have analyzed.

     More generally, the ARS paradigm does not elucidate the mechanisms which guide and **focus** a problem solver. ARS rule sets are capable of capturing only antecedent reasoning. Thus **all** deductions which can be made are made. This is only acceptable in domains which are self-**limiting** -- in which there are only a finite number of questions which might be asked and any of **them** might be relevent. Indeed, there is no means in ARS of focussing effort on the aspect of a **problem** which is being asked about. <u>Consequent reasoning</u> is one approach to model this part of **problem-solving** ability. More generally, rules to control the flow of reasoning may need the **antecedents** of a fact to be more complex -- one may want a fact to depend on the fact that another **fact** is false or even worse, unknown. Jon Doyle is now investigating a generalization of the ARS **dependency** scheme in which this is possible.

Appendix: An Annotated Example

This appendix documents a complete run of EL on the following circuit. It demonstrates how EL may be used not only to analyse a known circuit, but to help specify the component values to produce desired behavior.
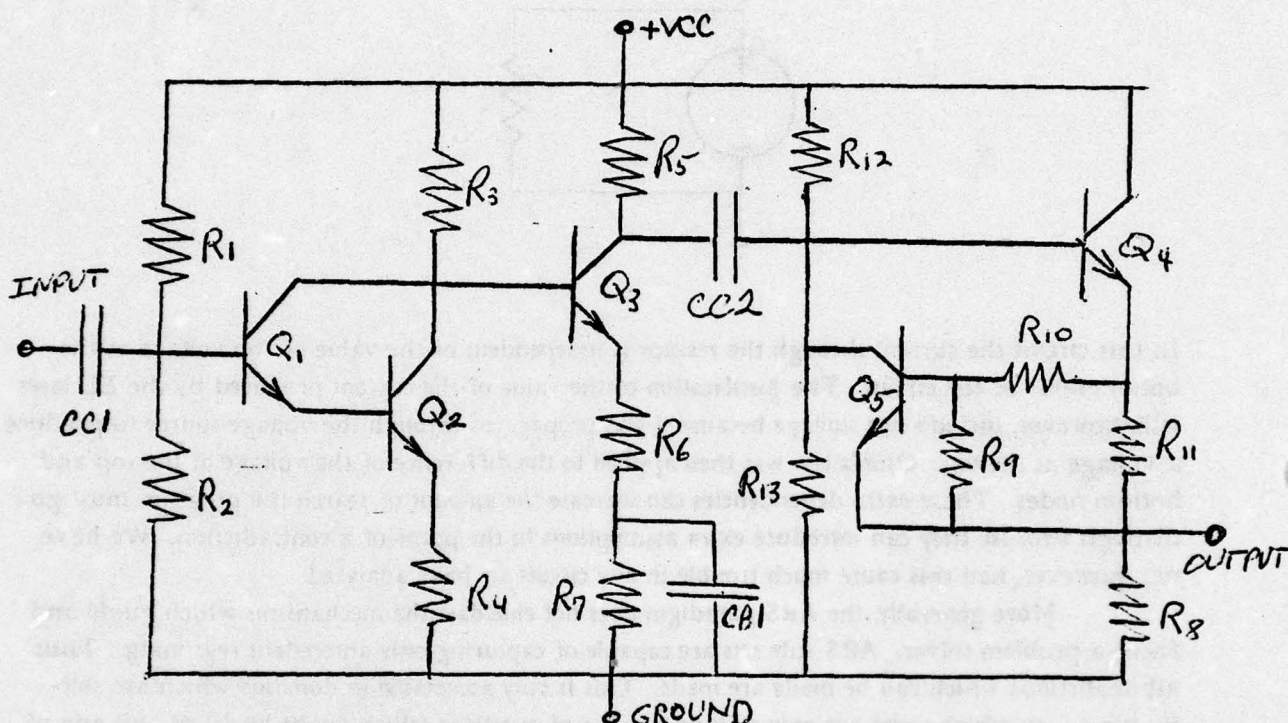
Our type-in is in lower case to distinguish it from ARS's output. The scenario is complete and un-cut. Do not hesitate to skim through long runs of ARS output not punctuated by English commentary; they contain more examples of types of reasoning that were already explained when they first appeared.

First we input the circuit diagram, specifying the name PROT2 for the circuit. We declare the names of the transistors, and specify their polarities and which semiconductor material they are made of. 0.6 indicates a silicon NPN transistor. Similarly, we give the names of all the resistors, and their resistances in ohms. Note that the name of a resistor doesn't *have* to begin with "R"; it is *just the standard electrical engineer's convention to do so.* Capacitors are specified with their values in farads. Currently the values are ignored, since analysis is done only for zero and infinite frequency. We then describe the interconnections of the devices. Each argument to CONNECT describes one node in the circuit by stating which terminals are incident with it. Each type of device has standard names for its terminals: for resistors and capacitors, "#1" and "#2"; for transistors, "E", "B" and "C". A terminal is a list whose second element is a device, and whose first

says which terminal of that device, as in (B Q1) for Q1's base. Certain nodes become "terminals" of the whole circuit, considered as a macro-device; they are given explicit names, such as GROUND, which identifies the node which will become the terminal (GROUND PROT2). PROT2 has four such terminals: GROUND, +VCC, INPUT and OUTPUT.

```
==> (wire prot2
      (bjt (q1 0.6) (q2 0.6) (q3 0.6) (q4 0.6) (q5 0.6))
      (resistor (r1 370000) (r2 100000) (r3 10000) (r4 1800) (r5 5600)
                (r6 630) (r7 3300) (r8 2200) (r9 1000) (r10 6800) (r11 470)
                (r12 39000) (r13 82000))
      (capacitor (cc1 2.0e-7) (cc2 1.0e-6) (cb1 1.0e-4))
      (connect (+vcc (#1 r1) (#1 r3) (#1 r5) (#1 r12) (c q4))
               (ground (#2 r2) (#2 r4) (#2 r7) (#2 cb1) (#2 r13) (#2 r8))
               (input (#1 cc1))
               ((#2 cc1) (#2 r1) (#1 r2) (b q1))
               ((#2 r3) (c q1) (c q2) (b q3))
               ((e q1) (b q2))
               ((e q2) (#1 r4))
               ((#2 r5) (c q3) (#1 cc2))
               ((#2 cc2) (#2 r12) (#1 r13) (b q4) (c q5))
               ((e q3) (#1 r6))
               ((#1 r7) (#2 r6) (#1 cb1))
               ((b q5) (#1 r9) (#1 r10))
               ((e q4) (#1 r11) (#2 r10))
               (output (e q5) (#2 r9) (#2 r11) (#1 r8))))
NIL
```

Now we supply boundary conditions for the circuit. ARS supplies a unique <u>factname</u> for each assertion we make, as it indexes the fact in the pattern-directed data base. The factname of an assertion is used for remembering its attributes or properties, such as how it was deduced, and what other assertions were deduced from it. Although an assertion could have an arbitrary form, the assertions meaningful to the EL system fall into a few schemata.

(= (VOLTAGE (GROUND PROT2)) 0.0) is an example of the most common schema of assertions in EL; it states that a particular parameter (VOLTAGE) at a particular terminal (namely, (GROUND PROT2)) is known to have a certain value (0.0).

VOLTAGE and CURRENT refer to the DC analysis of the circuit, while INC-VOLTS and INC-AMPS refer to the infinite-frequency analysis.

The first fact created is fact F226, because numbers 1 through 225 were used up by EL's initial knowledge (facts and demons), and by the assertions made during the wiring of the circuit.

```
==> (tell '(voltage (ground prot2)) 0)
USER: F226 (= (VOLTAGE (GROUND PROT2)) 0.0)
NIL
```

```
==> (tell '(voltage (+vcc prot2)) 15)
USER: F227 (= (VOLTAGE (+VCC PROT2)) 15.0)
NIL
==> (tell '(current (output prot2)) 0.0)
USER: F228 (= (CURRENT (OUTPUT PROT2)) 0.0)
NIL
==> (tell '(inc-volts (ground prot2)) 0)
USER: F229 (= (INC-VOLTS (GROUND PROT2)) 0.0)
NIL
==> (tell '(inc-volts (+vcc prot2)) 0)
USER: F230 (= (INC-VOLTS (+VCC PROT2)) 0.0)
NIL
==> (tell '(inc-amps (output prot2)) 0)
USER: F231 (= (INC-AMPS (OUTPUT PROT2)) 0.0)
NIL
==> (tell '(inc-volts (input prot2)) 0.1)
USER: F232 (= (INC-VOLTS (INPUT PROT2)) 0.1)
NIL
```

Not all the voltages and currents at the circuit's terminals have been specified; just enough to solve the circuit. The ones we did not give will be computed by EL as the circuit is solved. It might be surprising that only three DC boundary conditions are enough for a circuit which has four terminals, but they are because the capacitor CCl supplies what is effectively a fourth boundary condition: (= (CURRENT (INPUT PROT2)) 0.0).

The wiring of the circuit, and the setting of boundary conditions, have suggested to EL deductions to try to make. At this point we tell EL to follow them out as far as they go. Successful deductions give EL yet other suggestions. (RUN) will terminate only when EL has no idea how to deduce anything more.

Each deduction is made by a "law" which understands one particular *local* theorem or rule-of-thumb about a particular device type. Examples are Ohm's law for resistors, and the law that the base current of a transistor must be very small. Each deduction records the fact deduced, its factname, and the name of the law that deduced it. For example, DC-2T-SHORT is the law that the current into one end of a wire or "short" must equal the current out of the other end.

```
==> (run)
DC-CPROP: F233 (= (CURRENT (#1 W214)) 0.0)
DC-2T-SHORT: F234 (= (CURRENT (#2 W214)) 0.0)
DC-VPROP: F235 (= (VOLTAGE (#1 W133)) 15.0)
DC-SHORT: F236 (= (VOLTAGE (#2 W133)) 15.0)
DC-KVL: F237 (= (VOLTAGE (C Q4)) 15.0)
DC-KVL: F238 (= (VOLTAGE (#1 R12)) 15.0)
DC-KVL: F239 (= (VOLTAGE (#1 R5)) 15.0)
DC-KVL: F240 (= (VOLTAGE (#1 R3)) 15.0)
```

```
DC-KVL: F241 (= (VOLTAGE (#1 R1)) 15.0)
DC-VPROP: F242 (= (VOLTAGE (#1 W146)) 0.0)
DC-SHORT: F243 (= (VOLTAGE (#2 W146)) 0.0)
DC-KVL: F244 (= (VOLTAGE (#2 R8)) 0.0)
DC-KVL: F245 (= (VOLTAGE (#2 R13)) 0.0)
DC-KVL: F246 (= (VOLTAGE (#2 CB1)) 0.0)
DC-KVL: F247 (= (VOLTAGE (#2 R7)) 0.0)
DC-KVL: F248 (= (VOLTAGE (#2 R4)) 0.0)
DC-KVL: F249 (= (VOLTAGE (#2 R2)) 0.0)
DC-CAP-11: F250 (= (CURRENT (#1 CB1)) 0.0)
DC-CAP-12: F251 (= (CURRENT (#2 CB1)) 0.0)
DC-CAP-11: F252 (= (CURRENT (#1 CC2)) 0.0)
DC-CAP-12: F253 (= (CURRENT (#2 CC2)) 0.0)
DC-CAP-11: F254 (= (CURRENT (#1 CC1)) 0.0)
DC-CPROP: F255 (= (CURRENT (INPUT PROT2)) 0.0)
DC-CAP-12: F256 (= (CURRENT (#2 CC1)) 0.0)
```

The following NEEDCHOICE assertions drive the mechanism that chooses states for the transistors in the circuit, by triggering the demon that does the choosing. It is just a coincidence that these assertions happen now, when all the significant deductions have been exhausted; they might just as well have been the first assertions made, since the demons that they trigger have a special low priority that guarantees that they will not run until all deductions have been tried.

```
TRY-BJT: F257 (NEEDCHOICE (MODE Q5))
TRY-BJT: F258 (NEEDCHOICE (MODE Q4))
TRY-BJT: F259 (NEEDCHOICE (MODE Q3))
TRY-BJT: F260 (NEEDCHOICE (MODE Q2))
TRY-BJT: F261 (NEEDCHOICE (MODE Q1))
```

There are now no more one-step deductions available without assuming states for some of the transistors in the circuit. For the nonce, a transistor in EL has these states or "modes": "beta-infinite", "cutoff", and "saturated". The system will try choosing one mode and later try others instead if the first guess leads to a contradiction. After guessing the mode of one transistor. EL sees what can be deduced about the circuit before guessing the next. That is in the hope that some of the deductions will limit the possible choices for other transistors not yet guessed.

```
NEEDCHOICE: F262 (CHOICE (MODE Q1) BETA-INFINITE)
NEEDCHOICE: F263 (DETERMINED (MODE Q1) BETA-INFINITE)
DC-BJT-BETA-INFINITE: F264 (= (CURRENT (B Q1)) 6.0000001E-16)
NEEDCHOICE: F265 (CHOICE (MODE Q2) BETA-INFINITE)
NEEDCHOICE: F266 (DETERMINED (MODE Q2) BETA-INFINITE)
DC-BJT-BETA-INFINITE: F267 (= (CURRENT (B Q2)) 6.0000001E-16)
DC-KCL: F268 (= (CURRENT (E Q1)) -6.0000001E-16)
```

Think of UNDERFLOW as a very small number. It is our way of trying to avoid the problems caused by the nonassociativity of floating point addition. It is the result obtained when two large, nearly equal numbers have been subtracted and all precision lost. No contradiction results from equating UNDERFLOW to zero or to any small enough number. 6.0000001E-16, on the other hand, is a number which, while small compared to all the quantities of interest in the system, is nevertheless definitely *not* equal to zero. We use it to represent the small but certainly nonzero current into the base of an active transistor, so that a contradiction will occur if anything requires that it be exactly zero.

```
DC-KCL-BJT: F269 (= (CURRENT (C Q1)) UNDERFLOW)
NEEDCHOICE: F270 (CHOICE (MODE Q3) BETA-INFINITE)
NEEDCHOICE: F271 (DETERMINED (MODE Q3) BETA-INFINITE)
DC-BJT-BETA-INFINITE: F272 (= (CURRENT (B Q3)) 6.0000001E-16)
NEEDCHOICE: F273 (CHOICE (MODE Q4) BETA-INFINITE)
NEEDCHOICE: F274 (DETERMINED (MODE Q4) BETA-INFINITE)
DC-BJT-BETA-INFINITE: F275 (= (CURRENT (B Q4)) 6.0000001E-16)
NEEDCHOICE: F276 (CHOICE (MODE Q5) BETA-INFINITE)
NEEDCHOICE: F277 (DETERMINED (MODE Q5) BETA-INFINITE)
DC-BJT-BETA-INFINITE: F278 (= (CURRENT (B Q5)) 6.0000001E-16)
```

The current state of knowledge is:
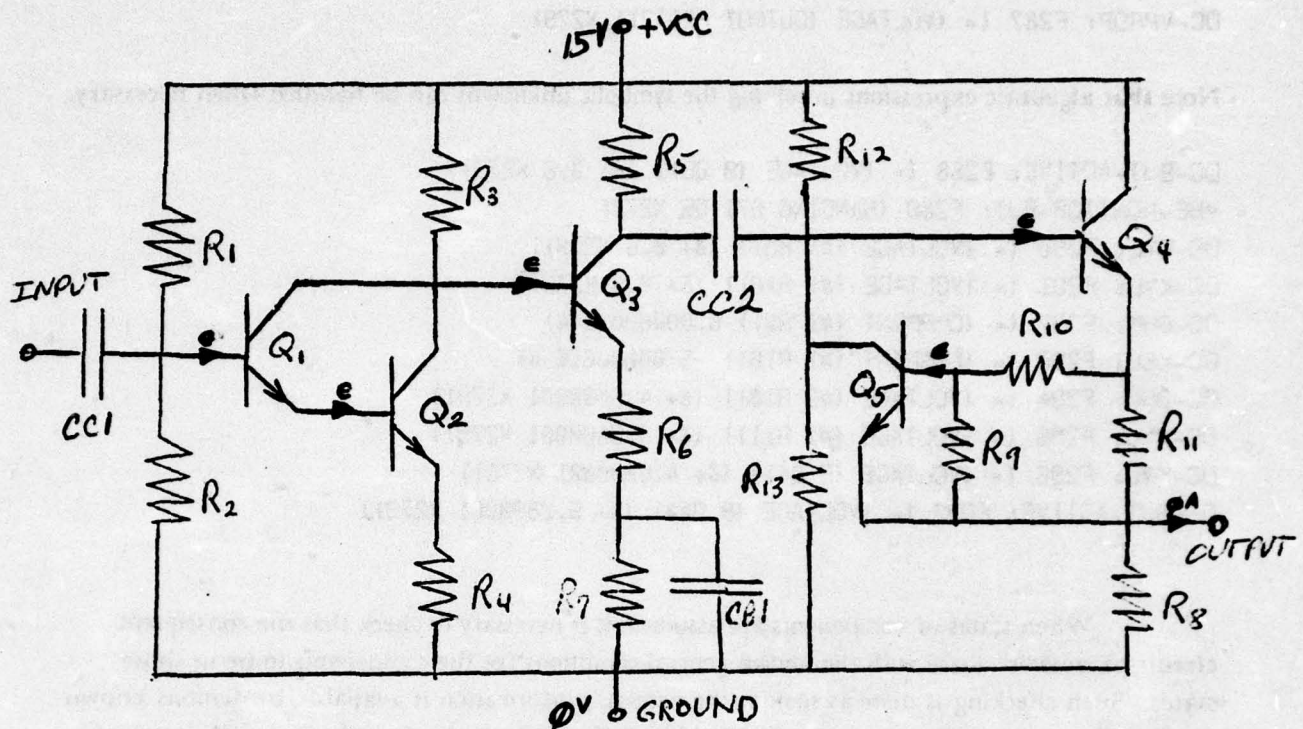
The system has again run out of one-step deductions. Since states have been assumed for all of the components that need them, the only possible cause is that the circuit contains an essential simultaneity of equations, and a symbolic unknown is necessary to proceed farther. The system creates a "symbolic unknown" X279, and assigns it as the value of one of the unknown voltages. This unknown can be propagated around via one-step deduction just as a numerical value can be. Eventually, when the "cycle" of simultaneous equations is "closed" by examining the last equation in it, all of its terms will reduce to expressions involving X279, whose value will then be determined (see assertion F323). From then on, whenever one of the previously propagated expressions involving X279 is referenced, X279 will be replaced with its value and the new expression will be simplified. Sometimes it happens that there are two coupled essential simultaneities; in that case The system will run out of things to do a second time before figuring out the value of the unknown. It then creates a second unknown and the two together would crack the simultaneity.

```
GENVARS: F280 (VARIABLE X279 (VOLTAGE (#1 R8)))
GENVARS: F281 (= (VOLTAGE (#1 R8)) X279)
DC-KVL: F282 (= (VOLTAGE (#2 R11)) X279)
DC-KVL: F283 (= (VOLTAGE (#2 R9)) X279)
DC-KVL: F284 (= (VOLTAGE (E Q5)) X279)
```

```
DC-KVL: F285 (= (VOLTAGE (#2 W214)) X279)
DC-SHORT: F286 (= (VOLTAGE (#1 W214)) X279)
DC-VPROP: F287 (= (VOLTAGE (OUTPUT PROT2)) X279)
```

Note that algebraic expressions involving the symbolic unknowns can be handled when necessary.

```
DC-BJT-ACTIVE: F288 (= (VOLTAGE (B Q5)) (&+ 0.6 X279))
VBE-MONITOR-BJT: F289 (HANGING D71 Q5 X279)
DC-KVL: F290 (= (VOLTAGE (#1 R9)) (&+ 0.6 X279))
DC-KVL: F291 (= (VOLTAGE (#1 R10)) (&+ 0.6 X279))
DC-OHM: F292 (= (CURRENT (#1 R9)) 6.0000001E-4)
DC-KCL: F293 (= (CURRENT (#1 R10)) -6.0000001E-4)
DC-OHM: F294 (= (VOLTAGE (#2 R10)) (&+ 4.6800001 X279))
DC-KVL: F295 (= (VOLTAGE (#1 R11)) (&+ 4.6800001 X279))
DC-KVL: F296 (= (VOLTAGE (E Q4)) (&+ 4.6800001 X279))
DC-BJT-ACTIVE: F297 (= (VOLTAGE (B Q4)) (&+ 5.2800001 X279))
```

When states of components are assumed, it is necessary to check that the consequent circuit parameters agree with the known general conditions for the components to be in those states. Such checking is done as soon as the necessary information is available, by demons known as <u>monitors</u>. One such monitor is VCB-MONITOR-BJT, which checks the voltage drop between the base and collector of a transistor for consistency. In this case, since the value just learned for the base voltage of Q4 involves an unknown, the monitor can't tell whether it is legitimate. It will be necessary to check again when the unknown's value becomes known. To that end, the monitor makes a HANGING assertion which associates that unknown with the need to apply this monitor, for this particular transistor. When the value is eventually learned, the HANGING assertion is noticed, and the monitor is tested once again with the new information.

```
VCB-MONITOR-BJT: F298 (HANGING D70 Q4 X279)
VBE-MONITOR-BJT: F299 (HANGING D71 Q4 X279)
DC-KVL: F300 (= (VOLTAGE (C Q5)) (&+ 5.2800001 X279))
DC-KVL: F301 (= (VOLTAGE (#1 R13)) (&+ 5.2800001 X279))
DC-KVL: F302 (= (VOLTAGE (#2 R12)) (&+ 5.2800001 X279))
DC-KVL: F303 (= (VOLTAGE (#2 CC2)) (&+ 5.2800001 X279))
VCB-MONITOR-BJT: F304 (HANGING D70 Q5 X279)
DC-OHM: F305 (= (CURRENT (#1 R12))
                (&+ 2.49230766E-4 (&* -2.56410256E-5 X279)))
DC-2T-R: F306 (= (CURRENT (#2 R12))
                (&+ -2.49230766E-4 (&* 2.56410256E-5 X279)))
DC-OHM: F307 (= (CURRENT (#1 R13))
                (&+ 6.4390245E-5 (&* 1.2195122E-5 X279)))
DC-KCL: F308 (= (CURRENT (C Q5)) (&+ 1.8484052E-4 (&* -3.7836147E-5 X279)))
```

```
IC-MONITOR-BJT: F309 (HANGING D66 Q5 X279)
NOTE-SATURATED: F310 (HANGING D72 Q5 X279)
DC-KCL-BJT: F311 (= (CURRENT (E Q5))
                        (&+ -1.8484052E-4 (&* 3.7836147E-5 X279)))
IE-MONITOR-BJT: F312 (HANGING D65 Q5 X279)
DC-2T-R: F313 (= (CURRENT (#2 R13)) (&+ -6.4390245E-5 (&* -1.2195122E-5 X279)))
DC-OHM: F314 (= (CURRENT (#1 R11)) 9.957447E-3)
DC-2T-R: F315 (= (CURRENT (#2 R11)) -9.957447E-3)
DC-2T-R: F316 (= (CURRENT (#2 R10)) 6.0000001E-4)
DC-KCL: F317 (= (CURRENT (E Q4)) -0.010557447)
DC-KCL-BJT: F318 (= (CURRENT (C Q4)) 0.010557447)
DC-2T-R: F319 (= (CURRENT (#2 R9)) -6.0000001E-4)
DC-KCL: F320 (= (CURRENT (#1 R8)) (&+ 0.0107422875 (&* -3.7836147E-5 X279)))
DC-OHM: F322 (CHECKED CHECK321 D1 R8)
DC-OHM: F323 (VALUE X279 21.816996)
```

Here we see that the value of the symbolic unknown X279 has finally been determined. This blessed event was the result of applying Ohm's law to four quantities each of which was already known in terms of X279. From now on, whenever a law makes use of a quantity previously dependent on X279, the known value of X279 will be substituted for it. When that happens, the final result is given an explicit dependency link to F323, since it would be invalid if X279's value ceased to be known. Such explicit dependency links were not necessary when X279 was propagated in symbolic form, since those propagations are valid regardless of the symbol's value. In addition, all of the monitor laws that were <u>hanging</u> on the value of X279 are now re-run.
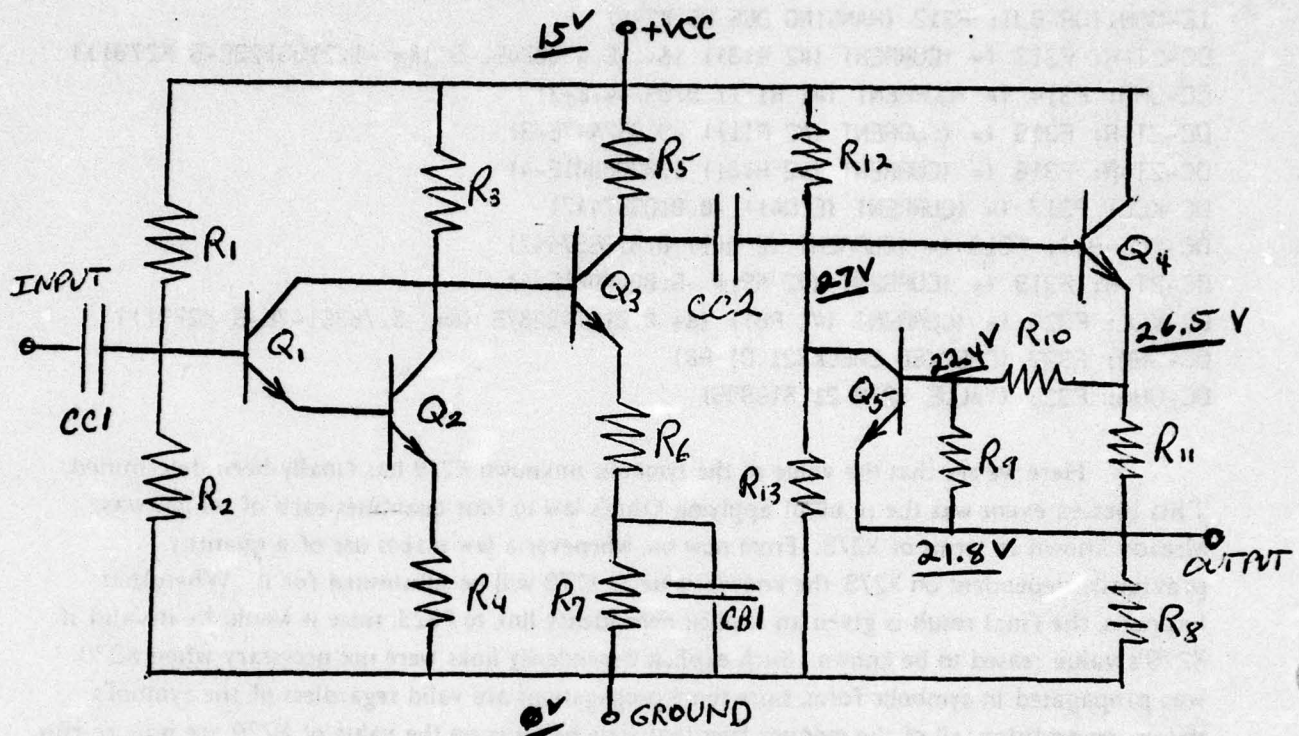
Unfortunately, our friend VCB-MONITOR-BJT detects an inconsistency: Q4's collector voltage is lower than its base voltage, even though Q4 is supposedly in the active state. When the contradiction is detected, the proof tree leading to it is searched for the component state choices which engendered it. Those choices, precisely, have been shown by the contradiction to be mutually inconsistent; A NOGOOD assertion is constructed to hold that insight and make sure that the same combination is never tried again. The fewer the number of choices in the NOGOOD, the more information has been gained from the contradiction. In this case there are only two choices in the NOGOOD. Of all the 3^5 (243) different sets of choices for all 5 transistors, there are 27 which include those two incompatible ones; all 27 have been ruled out at one blow. This phenomenon causes a great reduction in search space size, which makes EL converge in human timespans.

Once the system has determined the choices that are contradictory, it picks one of them to re-choose differently (the "culprit", actually chosen blindly). In this case, the culprit is F276 (Q5 active), the system "stops believing it" for the moment (but continues to remember what it would imply) and tries treating Q5 as cut off instead.

```
VCB-MONITOR-BJT: F325 (CONTRADICTION C324 D70 Q4)
CONTRADICTION: Q4 - F325 VCB-MONITOR-BJT
SUSPECTS:
F273  (CHOICE (MODE Q4) BETA-INFINITE)
```

F276   (CHOICE (MODE Q5) BETA-INFINITE)
CULPRIT: F276 unmarking; forgetting:
229 facts to check out
183 active facts left
1   contradictions so far
VCB-MONITOR-BJT: F326 (NOGOOD ((MODE Q4) BETA-INFINITE)
                                ((MODE Q5) BETA-INFINITE))
NEEDCHOICE: F327 (CHOICE (MODE Q5) CUTOFF)
NEEDCHOICE: F328 (DETERMINED (MODE Q5) CUTOFF)
DC-BJT-CUTOFF-IB: F329 (= (CURRENT (B Q5)) 0.0)
DC-BJT-CUTOFF-IC: F330 (= (CURRENT (C Q5)) 0.0)
DC-KCL-BJT: F331 (= (CURRENT (E Q5)) 0.0)

Despite the new state of one transistor, a symbolic unknown is still necessary. EL prefers
to use one already used when it is possible. As before, an assertion is made that a quantity has
that symbolic unknown as its value. But that assertion is identical to a previous, no-longer-
believed, assertion, F281. Therefore, instead of making a new assertion, the system once again
begins to believe F281. This facilitates _unouting_, a mechanism whereby all the old, never forgotten
but no longer believed, consequences of F281 are also restored to positions of trust. Unouting can
be viewed as a sort of cache for deductions: if it were not present, the same laws as applied the
previous time would succeed this time in deducing the same conclusions, but with much more
effort. Alternatively, it can be viewed as a context-switching mechanism which deals in trees of
"logically local" subcontexts.

DC-BJT-BETA-FINITE: F281 (= (VOLTAGE (#1 R8)) X279)
    Unout: F285 (= (VOLTAGE (#2 W214)) X279)
    Unout: F286 (= (VOLTAGE (#1 W214)) X279)
    Unout: F287 (= (VOLTAGE (OUTPUT PROT2)) X279)
    Unout: F284 (= (VOLTAGE (E Q5)) X279)
    Unout: F283 (= (VOLTAGE (#2 R9)) X279)
    Unout: F282 (= (VOLTAGE (#2 R11)) X279)
DC-OHM: F332 (= (CURRENT (#1 R8)) (&* 4.5454546E-4 X279))
DC-2T-R: F333 (= (CURRENT (#2 R8)) (&* -4.5454546E-4 X279))
GENVARS: F335 (VARIABLE X334 (VOLTAGE (#1 R9)))
GENVARS: F336 (= (VOLTAGE (#1 R9)) X334)
DC-KVL: F337 (= (VOLTAGE (B Q5)) X334)
DC-KVL: F338 (= (VOLTAGE (#1 R10)) X334)
VBE-MONITOR-BJT: F289 (HANGING D71 Q5 X279)
VBE-MONITOR-BJT: F339 (HANGING D71 Q5 X334)
DC-OHM: F340 (= (CURRENT (#1 R9)) (&+ (&* -1.0E-3 X279) (&* 1.0E-3 X334)))
DC-KCL: F341 (= (CURRENT (#1 R10)) (&+ (&* 1.0E-3 X279) (&* -1.0E-3 X334)))
DC-OHM: F342 (= (VOLTAGE (#2 R10)) (&+ (&* -6.8 X279) (&* 7.8 X334)))
DC-KVL: F343 (= (VOLTAGE (#1 R11)) (&+ (&* -6.8 X279) (&* 7.8 X334)))

DC-KVL: F344 (= (VOLTAGE (E Q4)) (&+ (&* -6.8 X279) (&* 7.8 X334)))
DC-BJT-ACTIVE: F345 (= (VOLTAGE (B Q4)) (&+ 0.6 (&* -6.8 X279) (&* 7.8 X334)))
VCB-MONITOR-BJT: F346 (HANGING D70 Q4 X334)
VCB-MONITOR-BJT: F298 (HANGING D70 Q4 X279)
VBE-MONITOR-BJT: F347 (HANGING D71 Q4 X334)
VBE-MONITOR-BJT: F299 (HANGING D71 Q4 X279)
DC-KVL: F348 (= (VOLTAGE (C Q5)) (&+ 0.6 (&* -6.8 X279) (&* 7.8 X334)))
DC-KVL: F349 (= (VOLTAGE (#1 R13)) (&+ 0.6 (&* -6.8 X279) (&* 7.8 X334)))
DC-KVL: F350 (= (VOLTAGE (#2 R12)) (&+ 0.6 (&* -6.8 X279) (&* 7.8 X334)))
DC-KVL: F351 (= (VOLTAGE (#2 CC2)) (&+ 0.6 (&* -6.8 X279) (&* 7.8 X334)))
VCB-MONITOR-BJT: F304 (HANGING D70 Q5 X279)
VCB-MONITOR-BJT: F352 (HANGING D70 Q5 X334)
DC-OHM: F353 (= (CURRENT (#1 R12))
                  (&+ 3.6923077E-4 (&* 1.74358974E-4 X279) (&* -2.0E-4 X334)))
DC-2T-R: F354 (= (CURRENT (#2 R12))
                  (&+ -3.6923077E-4 (&* -1.74358974E-4 X279) (&* 2.0E-4 X334)))
DC-KCL: F355 (= (CURRENT (#1 R13))
                  (&+ 3.6923077E-4 (&* 1.74358974E-4 X279) (&* -2.0E-4 X334)))
DC-OHM: F357 (CHECKED CHECK356 D1 R13)
DC-OHM: F358 (VALUE X334 (&+ 1.22631915 (&* 0.87179488 X279)))
DC-2T-R: F359 (= (CURRENT (#2 R13)) -1.2396694E-4)
DC-OHM: F360 (= (CURRENT (#1 R11)) (&+ 0.0203516795 (&* -2.12765956E-3 X279)))
DC-2T-R: F361 (= (CURRENT (#2 R11)) (&+ -0.0203516795 (&* 2.12765956E-3 X279)))
DC-KCL: F362 (= (CURRENT (#2 R9)) (&+ 0.0203516795 (&* -2.582205E-3 X279)))
DC-2T-R: F364 (CHECKED CHECK363 D2 R9)
DC-2T-R: F365 (VALUE X279 7.9611562)
DC-2T-R: F366 (= (CURRENT (#2 R10)) 2.05658144E-4)
DC-KCL: F367 (= (CURRENT (E Q4)) -3.6187074E-3)
DC-KCL-BJT: F368 (= (CURRENT (C Q4)) 3.6187074E-3)
GENVARS: F370 (VARIABLE X369 (VOLTAGE (#1 CB1)))
GENVARS: F371 (= (VOLTAGE (#1 CB1)) X369)
DC-KVL: F372 (= (VOLTAGE (#2 R6)) X369)
DC-KVL: F373 (= (VOLTAGE (#1 R7)) X369)
DC-OHM: F374 (= (CURRENT (#1 R7)) (&* 3.030303E-4 X369))
DC-KCL: F375 (= (CURRENT (#2 R6)) (&* -3.030303E-4 X369))
DC-2T-R: F376 (= (CURRENT (#1 R6)) (&* 3.030303E-4 X369))
DC-KCL: F377 (= (CURRENT (E Q3)) (&* -3.030303E-4 X369))
IE-MONITOR-BJT: F378 (HANGING D65 Q3 X369)
DC-KCL-BJT: F379 (= (CURRENT (C Q3)) (&+ -6.0000001E-16 (&* 3.030303E-4 X369)))
IC-MONITOR-BJT: F380 (HANGING D66 Q3 X369)
NOTE-SATURATED: F381 (HANGING D72 Q3 X369)
DC-KCL: F382 (= (CURRENT (#2 R5)) (&+ 6.0000001E-16 (&* -3.030303E-4 X369)))
DC-2T-R: F383 (= (CURRENT (#1 R5)) (&+ -6.0000001E-16 (&* 3.030303E-4 X369)))

DC-OHM: F384 (= (VOLTAGE (#2 R5)) (&+ 15.0 (&* -1.6969697 X369)))
DC-KVL: F385 (= (VOLTAGE (#1 CC2)) (&+ 15.0 (&* -1.6969697 X369)))
DC-KVL: F386 (= (VOLTAGE (C Q3)) (&+ 15.0 (&* -1.6969697 X369)))
DC-OHM: F387 (= (VOLTAGE (#1 R6)) (&* 1.1909091 X369))
DC-KVL: F388 (= (VOLTAGE (E Q3)) (&* 1.1909091 X369))
DC-BJT-ACTIVE: F389 (= (VOLTAGE (B Q3)) (&+ 0.6 (&* 1.1909091 X369)))
VCB-MONITOR-BJT: F390 (HANGING D70 Q3 X369)
VBE-MONITOR-BJT: F391 (HANGING D71 Q3 X369)
DC-KVL: F392 (= (VOLTAGE (C Q2)) (&+ 0.6 (&* 1.1909091 X369)))
DC-KVL: F393 (= (VOLTAGE (C Q1)) (&+ 0.6 (&* 1.1909091 X369)))
DC-KVL: F394 (= (VOLTAGE (#2 R3)) (&+ 0.6 (&* 1.1909091 X369)))
DC-OHM: F395 (= (CURRENT (#1 R3)) (&+ 1.44E-3 (&* -1.1909091E-4 X369)))
DC-2T-R: F396 (= (CURRENT (#2 R3)) (&+ -1.44E-3 (&* 1.1909091E-4 X369)))
DC-KCL: F397 (= (CURRENT (C Q2)) (&+ 1.44E-3 (&* -1.1909091E-4 X369)))
IC-MONITOR-BJT: F398 (HANGING D66 Q2 X369)
NOTE-SATURATED: F399 (HANGING D72 Q2 X369)
DC-KCL-BJT: F400 (= (CURRENT (E Q2)) (&+ -1.44E-3 (&* 1.1909091E-4 X369)))
IE-MONITOR-BJT: F401 (HANGING D65 Q2 X369)
DC-KCL: F402 (= (CURRENT (#1 R4)) (&+ 1.44E-3 (&* -1.1909091E-4 X369)))
DC-OHM: F403 (= (VOLTAGE (#1 R4)) (&+ 2.592 (&* -0.214363636 X369)))
DC-KVL: F404 (= (VOLTAGE (E Q2)) (&+ 2.592 (&* -0.214363636 X369)))
DC-BJT-ACTIVE: F405 (= (VOLTAGE (B Q2)) (&+ 3.192 (&* -0.214363636 X369)))
VCB-MONITOR-BJT: F406 (HANGING D70 Q2 X369)
VBE-MONITOR-BJT: F407 (HANGING D71 Q2 X369)
DC-KVL: F408 (= (VOLTAGE (E Q1)) (&+ 3.192 (&* -0.214363636 X369)))
DC-BJT-ACTIVE: F409 (= (VOLTAGE (B Q1)) (&+ 3.792 (&* -0.214363636 X369)))
VCB-MONITOR-BJT: F410 (HANGING D70 Q1 X369)
VBE-MONITOR-BJT: F411 (HANGING D71 Q1 X369)
DC-KVL: F412 (= (VOLTAGE (#1 R2)) (&+ 3.792 (&* -0.214363636 X369)))
DC-KVL: F413 (= (VOLTAGE (#2 R1)) (&+ 3.792 (&* -0.214363636 X369)))
DC-KVL: F414 (= (VOLTAGE (#2 CC1)) (&+ 3.792 (&* -0.214363636 X369)))
DC-OHM: F415 (= (CURRENT (#1 R1)) (&+ 3.0291892E-5 (&* 5.7936118E-7 X369)))
DC-KCL: F416 (= (CURRENT (#2 W133))
                  (&+ -5.2129663E-3 (&* -1.84518754E-4 X369)))
DC-2T-SHORT: F417 (= (CURRENT (#1 W133))
                     (&+ 5.2129663E-3 (&* 1.84518754E-4 X369)))
DC-CPROP: F418 (= (CURRENT (+VCC PROT2))
                  (&+ 5.2129663E-3 (&* 1.84518754E-4 X369)))
DC-2T-R: F419 (= (CURRENT (#2 R1)) (&+ -3.0291892E-5 (&* -5.7936118E-7 X369)))
DC-KCL: F420 (= (CURRENT (#1 R2)) (&+ 3.0291892E-5 (&* 5.7936118E-7 X369)))
DC-OHM: F422 (CHECKED CHECK421 D1 R2)
DC-OHM: F423 (VALUE X369 2.8013641)
DC-2T-R: F424 (= (CURRENT (#2 R2)) -3.1914894E-5)

```
DC-2T-R: F425 (= (CURRENT (#2 R4)) -1.106383E-3)
DC-2T-R: F426 (= (CURRENT (#2 R7)) -8.488982E-4)
DC-KCL: F427 (= (CURRENT (#2 W146)) 5.7298704E-3)
DC-2T-SHORT: F428 (= (CURRENT (#1 W146)) -5.7298704E-3)
DC-CPROP: F429 (= (CURRENT (GROUND PROT2)) -5.7298704E-3)
```

Now all of the DC parameters of the circuit have been determined. At this time, EL begins to pay attention to the AC parameters. It would be perfectly possible to deal with AC and DC analysis indiscriminately from the beginning, but is desirable to do as little work that will be forgotten due to contradictions as is possible, and therefore whatever is likely to produce a contradiction (such as DC analysis) should receive higher priority than what will not produce one (such as AC analysis), assuming that both will have to be done in the end.

```
INC-VPROP: F430 (= (INC-VOLTS (#1 CC1)) 0.1)
INC-CAP: F431 (= (INC-VOLTS (#2 CC1)) 0.1)
INC-KVL: F432 (= (INC-VOLTS (B Q1)) 0.1)
INC-KVL: F433 (= (INC-VOLTS (#1 R2)) 0.1)
INC-KVL: F434 (= (INC-VOLTS (#2 R1)) 0.1)
INC-BJT-ACTIVE: F435 (= (INC-VOLTS (E Q1)) 0.1)
INC-KVL: F436 (= (INC-VOLTS (B Q2)) 0.1)
INC-BJT-ACTIVE: F437 (= (INC-VOLTS (E Q2)) 0.1)
INC-KVL: F438 (= (INC-VOLTS (#1 R4)) 0.1)
INC-CPROP: F439 (= (INC-AMPS (#1 W214)) 0.0)
INC-2T-SHORT: F440 (= (INC-AMPS (#2 W214)) 0.0)
INC-VPROP: F441 (= (INC-VOLTS (#1 W133)) 0.0)
INC-SHORT: F442 (= (INC-VOLTS (#2 W133)) 0.0)
INC-KVL: F443 (= (INC-VOLTS (C Q4)) 0.0)
INC-KVL: F444 (= (INC-VOLTS (#1 R12)) 0.0)
INC-KVL: F445 (= (INC-VOLTS (#1 R5)) 0.0)
INC-KVL: F446 (= (INC-VOLTS (#1 R3)) 0.0)
INC-KVL: F447 (= (INC-VOLTS (#1 R1)) 0.0)
INC-OHM: F448 (= (INC-AMPS (#1 R1)) -2.7027027E-7)
INC-2T-R: F449 (= (INC-AMPS (#2 R1)) 2.7027027E-7)
INC-VPROP: F450 (= (INC-VOLTS (#1 W146)) 0.0)
INC-SHORT: F451 (= (INC-VOLTS (#2 W146)) 0.0)
INC-KVL: F452 (= (INC-VOLTS (#2 R8)) 0.0)
INC-KVL: F453 (= (INC-VOLTS (#2 R13)) 0.0)
INC-KVL: F454 (= (INC-VOLTS (#2 CB1)) 0.0)
INC-KVL: F455 (= (INC-VOLTS (#2 R7)) 0.0)
INC-KVL: F456 (= (INC-VOLTS (#2 R4)) 0.0)
INC-KVL: F457 (= (INC-VOLTS (#2 R2)) 0.0)
INC-OHM: F458 (= (INC-AMPS (#1 R2)) 1.0E-6)
INC-2T-R: F459 (= (INC-AMPS (#2 R2)) -1.0E-6)
```

```
INC-OHM: F460 (= (INC-AMPS (#1 R4)) 5.5555555E-5)
INC-KCL: F461 (= (INC-AMPS (E Q2)) -5.5555555E-5)
INC-2T-R: F462 (= (INC-AMPS (#2 R4)) -5.5555555E-5)
INC-CAP: F463 (= (INC-VOLTS (#1 CB1)) 0.0)
INC-KVL: F464 (= (INC-VOLTS (#2 R6)) 0.0)
INC-KVL: F465 (= (INC-VOLTS (#1 R7)) 0.0)
INC-OHM: F466 (= (INC-AMPS (#1 R7)) 0.0)
INC-2T-R: F467 (= (INC-AMPS (#2 R7)) 0.0)
INC-BJT-CUTOFF-IB: F468 (= (INC-AMPS (B Q5)) 0.0)
INC-BJT-CUTOFF-IC: F469 (= (INC-AMPS (C Q5)) 0.0)
INC-KCL-BJT: F470 (= (INC-AMPS (E Q5)) 0.0)
INC-BJT-BETA-INFINITE: F471 (= (INC-AMPS (B Q4)) 0.0)
INC-BJT-BETA-INFINITE: F472 (= (INC-AMPS (B Q3)) 0.0)
INC-BJT-BETA-INFINITE: F473 (= (INC-AMPS (B Q2)) 0.0)
INC-KCL: F474 (= (INC-AMPS (E Q1)) 0.0)
INC-KCL-BJT: F475 (= (INC-AMPS (C Q2)) 5.5555555E-5)
INC-BJT-BETA-INFINITE: F476 (= (INC-AMPS (B Q1)) 0.0)
INC-KCL: F477 (= (INC-AMPS (#2 CC1)) -1.27027027E-6)
INC-2T-C: F478 (= (INC-AMPS (#1 CC1)) 1.27027027E-6)
INC-CPROP: F479 (= (INC-AMPS (INPUT PROT2)) 1.27027027E-6)
INC-KCL-BJT: F480 (= (INC-AMPS (C Q1)) 0.0)
INC-KCL: F481 (= (INC-AMPS (#2 R3)) -5.5555555E-5)
INC-2T-R: F482 (= (INC-AMPS (#1 R3)) 5.5555555E-5)
INC-OHM: F483 (= (INC-VOLTS (#2 R3)) -0.55555555)
INC-KVL: F484 (= (INC-VOLTS (B Q3)) -0.55555555)
INC-KVL: F485 (= (INC-VOLTS (C Q2)) -0.55555555)
INC-KVL: F486 (= (INC-VOLTS (C Q1)) -0.55555555)
INC-BJT-ACTIVE: F487 (= (INC-VOLTS (E Q3)) -0.55555555)
INC-KVL: F488 (= (INC-VOLTS (#1 R6)) -0.55555555)
INC-OHM: F489 (= (INC-AMPS (#1 R6)) -8.818342E-4)
INC-KCL: F490 (= (INC-AMPS (E Q3)) 8.818342E-4)
INC-KCL-BJT: F491 (= (INC-AMPS (C Q3)) -8.818342E-4)
INC-2T-R: F492 (= (INC-AMPS (#2 R6)) 8.818342E-4)
INC-KCL: F493 (= (INC-AMPS (#1 CB1)) -8.818342E-4)
INC-2T-C: F494 (= (INC-AMPS (#2 CB1)) 8.818342E-4)
GENVARS: F496 (VARIABLE X495 (INC-VOLTS (#1 R8)))
GENVARS: F497 (= (INC-VOLTS (#1 R8)) X495)
INC-KVL: F498 (= (INC-VOLTS (#2 R11)) X495)
INC-KVL: F499 (= (INC-VOLTS (#2 R9)) X495)
INC-KVL: F500 (= (INC-VOLTS (E Q5)) X495)
INC-KVL: F501 (= (INC-VOLTS (#2 W214)) X495)
INC-SHORT: F502 (= (INC-VOLTS (#1 W214)) X495)
INC-VPROP: F503 (= (INC-VOLTS (OUTPUT PROT2)) X495)
```

```
INC-OHM: F504 (= (INC-AMPS (#1 R8)) (&* 4.5454546E-4 X495))
INC-2T-R: F505 (= (INC-AMPS (#2 R8)) (&* -4.5454546E-4 X495))
GENVARS: F507 (VARIABLE X506 (INC-VOLTS (#1 R9)))
GENVARS: F508 (= (INC-VOLTS (#1 R9)) X506)
INC-KVL: F509 (= (INC-VOLTS (B Q5)) X506)
INC-KVL: F510 (= (INC-VOLTS (#1 R10)) X506)
INC-OHM: F511 (= (INC-AMPS (#1 R9)) (&+ (&* -1.0E-3 X495) (&* 1.0E-3 X506)))
INC-KCL: F512 (= (INC-AMPS (#1 R10)) (&+ (&* 1.0E-3 X495) (&* -1.0E-3 X506)))
INC-OHM: F513 (= (INC-VOLTS (#2 R10)) (&+ (&* -6.8 X495) (&* 7.8 X506)))
INC-KVL: F514 (= (INC-VOLTS (#1 R11)) (&+ (&* -6.8 X495) (&* 7.8 X506)))
INC-KVL: F515 (= (INC-VOLTS (E Q4)) (&+ (&* -6.8 X495) (&* 7.8 X506)))
INC-BJT-ACTIVE: F516 (= (INC-VOLTS (B Q4)) (&+ (&* -6.8 X495) (&* 7.8 X506)))
INC-KVL: F517 (= (INC-VOLTS (C Q5)) (&+ (&* -6.8 X495) (&* 7.8 X506)))
INC-KVL: F518 (= (INC-VOLTS (#1 R13)) (&+ (&* -6.8 X495) (&* 7.8 X506)))
INC-KVL: F519 (= (INC-VOLTS (#2 R12)) (&+ (&* -6.8 X495) (&* 7.8 X506)))
INC-KVL: F520 (= (INC-VOLTS (#2 CC2)) (&+ (&* -6.8 X495) (&* 7.8 X506)))
INC-CAP: F521 (= (INC-VOLTS (#1 CC2)) (&+ (&* -6.8 X495) (&* 7.8 X506)))
INC-KVL: F522 (= (INC-VOLTS (C Q3)) (&+ (&* -6.8 X495) (&* 7.8 X506)))
INC-KVL: F523 (= (INC-VOLTS (#2 R5)) (&+ (&* -6.8 X495) (&* 7.8 X506)))
INC-OHM: F524 (= (INC-AMPS (#1 R5))
                    (&+ (&* 1.21428571E-3 X495) (&* -1.39285715E-3 X506)))
INC-2T-R: F525 (= (INC-AMPS (#2 R5))
                    (&+ (&* -1.21428571E-3 X495) (&* 1.39285715E-3 X506)))
INC-KCL: F526 (= (INC-AMPS (#1 CC2))
                    (&+ 8.818342E-4 (&* 1.21428571E-3 X495) (&* -1.39285715E-3 X506)))
INC-2T-C: F527 (= (INC-AMPS (#2 CC2))
                    (&+ -8.818342E-4 (&* -1.21428571E-3 X495) (&* 1.39285715E-3 X506)))
INC-OHM: F528 (= (INC-AMPS (#1 R12)) (&+ (&* 1.74358974E-4 X495) (&* -2.0E-4 X506)))
INC-2T-R: F529 (= (INC-AMPS (#2 R12)) (&+ (&* -1.74358974E-4 X495) (&* 2.0E-4 X506)))
INC-KCL: F530 (= (INC-AMPS (#1 R13))
                    (&+ 8.818342E-4 (&* 1.3886447E-3 X495) (&* -1.59285713E-3 X506)))
INC-OHM: F532 (CHECKED CHECK531 D3 R13)
INC-OHM: F533 (VALUE X506 (&+ 0.52242011 (&* 0.87179488 X495)))
INC-2T-R: F534 (= (INC-AMPS (#2 R13)) -4.9693612E-5)
INC-KCL: F535 (= (INC-AMPS (#2 W146)) (&+ -7.7558504E-4 (&* 4.5454546E-4 X495)))
INC-2T-SHORT: F536 (= (INC-AMPS (#1 W146)) (&+ 7.7558504E-4 (&* -4.5454546E-4 X495)))
INC-CPROP: F537 (= (INC-AMPS (GROUND PROT2)) (&+ 7.7558504E-4 (&* -4.5454546E-4 X495)))
INC-OHM: F538 (= (INC-AMPS (#1 R11)) (&+ 8.6699506E-3 (&* -2.12765956E-3 X495)))
INC-2T-R: F539 (= (INC-AMPS (#2 R11)) (&+ -8.6699506E-3 (&* 2.12765956E-3 X495)))
INC-KCL: F540 (= (INC-AMPS (#2 R9)) (&+ 8.6699506E-3 (&* -2.582205E-3 X495)))
INC-2T-R: F542 (CHECKED CHECK541 D4 R9)
INC-2T-R: F543 (VALUE X495 3.3915055)
INC-2T-R: F544 (= (INC-AMPS (#2 R10)) 8.7611785E-5)
```

INC-KCL: F545 (= (INC-AMPS (E Q4)) -1.54159334E-3)
INC-KCL-BJT: F546 (= (INC-AMPS (C Q4)) 1.54159334E-3)
INC-KCL: F547 (= (INC-AMPS (#2 W133)) -7.6473795E-4)
INC-2T-SHORT: F548 (= (INC-AMPS (#1 W133)) 7.6473795E-4)
INC-CPROP: F549 (= (INC-AMPS (+VCC PROT2)) 7.6473795E-4)
DONE
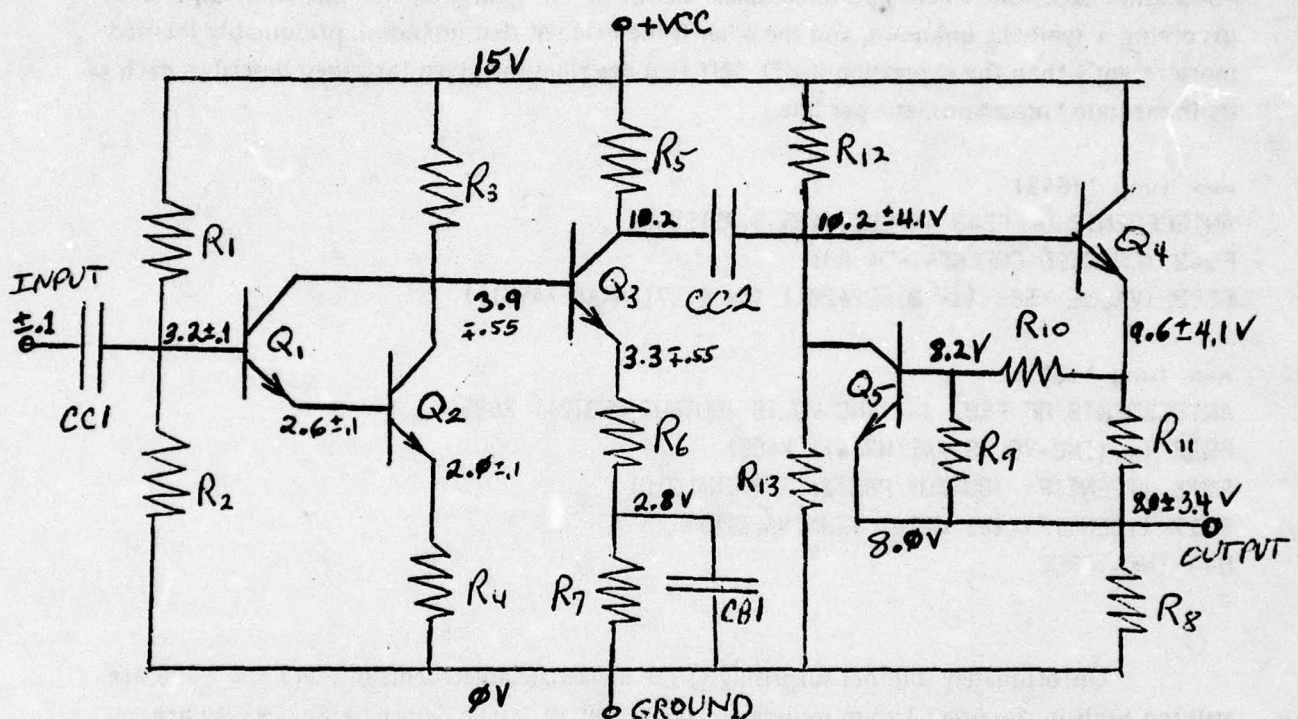
The analysis is complete! We can now ask for the values of the quantities of interest to us. The process of asking replaces all symbolic "unknowns" of known value with their values, and performs arithmetic operations, to make the answer more transparent. It also lists the factnames facts accessed in finding the answer.

==> (what '(voltage (output prot2)))
7.9611562 (F365 F287)
T
==> (what '(inc-volts (output prot2)))
3.3915055 (F543 F503)
T

Since the input INC-VOLTS was .1, the output of 3.4 indicates a gain of 34 for the circuit.

Suppose that we, the circuit designers, are unsatisfied with the gain, as expressed by those results, and would like to alter the circuit device parameters to make the gain exactly 40. We can use EL to do this; first, we would like to find out which device parameters can be used to vary the gain without changing the output bias level, and vice versa.

We use the fact that we can ask ARS how it deduced any fact that it knows. WHY prints only the immediate antecedents; EXPLAIN prints the entire proof. We use them on the two facts, F543 and F503, which went into the ultimate answer for the gain (note that one is an expression involving a symbolic unknown, and the other is the value of that unknown, presumably learned more recently than the expression itself). WHY first describes the given fact, then describes each of its immediate antecedents, one per line.

```
==> (why 'f543)
ANTECEDENTS OF F543 (VALUE X495 3.3915055)
F542 (CHECKED CHECK541 D4 R9)
F533 (VALUE X506 (&+ 0.52242011 (&* 0.87179488 X495)))
?
==> (why 'f503)
ANTECEDENTS OF F503 (= (INC-VOLTS (OUTPUT PROT2)) X495)
F502 (= (INC-VOLTS (#1 W214)) X495)
F218 (IDENTIFY (OUTPUT PROT2) TERMINAL216)
F217 (IDENTIFY (#1 W214) TERMINAL216)
D44 INC-VPROP
?
```

Unfortunately (but not surprisingly), the immediate antecedents of F543 and F503 are still too high in the proof tree to include any of the initially-given device-parameters we are looking for, so we must examine the entire proof. For each fact that appears in the proof, EXPLAIN prints the factname and then the statement of the fact, followed by the factnames of the fact's antecedents. The antecedents' factnames may be interpreted by looking for them on other lines of the printed proof, which is ordered numerically by decreasing factnames. The antecedent facts will usually be accompanied by the name of the demon which performed the deduction. GIVEN as an "antecedent" indicates an assertion that was assumed or given by the user, rather than deduced. USER as well indicates a fact specified interactively by the user with the TELL function.

The only parts of the proof that we are actually interested in are the facts that were "axioms" specified when the circuit was wired up. They come near the end because their factnames are all low-numbered.

```
==> (explain 'f543)
F543 (VALUE X495 3.3915055) (F542 F533)
F542 (CHECKED CHECK541 D4 R9) (F540 F511 INC-2T-R)
F540 (= (INC-AMPS (#2 R9))
```

```
                (&+ 8.6699506E-3 (&* -2.582205E-3 X495))))
        (F539 F504 F470 F440 INC-KCL)
F539 (= (INC-AMPS (#2 R11))) (&+ -8.6699506E-3 (&* 2.12765956E-3 X495))) (F538 INC-2T-R)
F538 (= (INC-AMPS (#1 R11)))
        (&+ 8.6699506E-3 (&* -2.12765956E-3 X495))))
        (F533 F514 F498 F122 INC-OHM)
F533 (VALUE X506 (&+ 0.52242011 (&* 0.87179488 X495))) (F532)
F532 (CHECKED CHECK531 D3 R13) (F530 F518 F453 F126 INC-OHM)
F530 (= (INC-AMPS (#1 R13)))
        (&+ 8.818342E-4 (&* 1.3886447E-3 X495) (&* -1.59285713E-3 X506)))
        (F529 F527 F471 F469 INC-KCL)
F529 (= (INC-AMPS (#2 R12)))
        (&+ (&* -1.74358974E-4 X495) (&* 2.0E-4 X506)))
        (F528 INC-2T-R)
F528 (= (INC-AMPS (#1 R12)))
        (&+ (&* 1.74358974E-4 X495) (&* -2.0E-4 X506)))
        (F519 F444 F124 INC-OHM)
F527 (= (INC-AMPS (#2 CC2)))
        (&+ -8.818342E-4 (&* -1.21428571E-3 X495) (&* 1.39285715E-3 X506)))
        (F526 INC-2T-C)
F526 (= (INC-AMPS (#1 CC2)))
        (&+ 8.818342E-4 (&* 1.21428571E-3 X495) (&* -1.39285715E-3 X506)))
        (F525 F491 INC-KCL)
F525 (= (INC-AMPS (#2 R5)))
        (&+ (&* -1.21428571E-3 X495) (&* 1.39285715E-3 X506)))
        (F524 INC-2T-R)
F524 (= (INC-AMPS (#1 R5)))
        (&+ (&* 1.21428571E-3 X495) (&* -1.39285715E-3 X506)))
        (F523 F445 F110 INC-OHM)
F523 (= (INC-VOLTS (#2 R5)) (&+ (&* -6.8 X495) (&* 7.8 X506))) (F521 INC-KVL)
F521 (= (INC-VOLTS (#1 CC2)) (&+ (&* -6.8 X495) (&* 7.8 X506))) (F520 INC-CAP)
F520 (= (INC-VOLTS (#2 CC2)) (&+ (&* -6.8 X495) (&* 7.8 X506))) (F516 INC-KVL)
F519 (= (INC-VOLTS (#2 R12)) (&+ (&* -6.8 X495) (&* 7.8 X506))) (F516 INC-KVL)
F518 (= (INC-VOLTS (#1 R13)) (&+ (&* -6.8 X495) (&* 7.8 X506))) (F516 INC-KVL)
F516 (= (INC-VOLTS (B Q4)) (&+ (&* -6.8 X495) (&* 7.8 X506))) (F515 F274 INC-BJT-ACTIVE)
F515 (= (INC-VOLTS (E Q4)) (&+ (&* -6.8 X495) (&* 7.8 X506))) (F513 INC-KVL)
F514 (= (INC-VOLTS (#1 R11)) (&+ (&* -6.8 X495) (&* 7.8 X506))) (F513 INC-KVL)
F513 (= (INC-VOLTS (#2 R10)) (&+ (&* -6.8 X495) (&* 7.8 X506))) (F512 F510 F120 INC-OHM)
F512 (= (INC-AMPS (#1 R10)) (&+ (&* 1.0E-3 X495) (&* -1.0E-3 X506))) (F511 F468 INC-KCL)
F511 (= (INC-AMPS (#1 R9)) (&+ (&* -1.0E-3 X495) (&* 1.0E-3 X506)))
        (F508 F499 F118 INC-OHM)
F510 (= (INC-VOLTS (#1 R10)) X506) (F508 INC-KVL)
F508 (= (INC-VOLTS (#1 R9)) X506) (GIVEN F327 F273 F270 F265 F262)
```

```
F504 (= (INC-AMPS (#1 R8)) (&* 4.5454546E-4 X495)) (F497 F452 F116 INC-OHM)
F499 (= (INC-VOLTS (#2 R9)) X495) (F497 INC-KVL)
F498 (= (INC-VOLTS (#2 R11)) X495) (F497 INC-KVL)
F497 (= (INC-VOLTS (#1 R8)) X495) (GIVEN F327 F273 F270 F265 F262)
F491 (= (INC-AMPS (C Q3)) -8.818342E-4) (F490 F472 INC-KCL-BJT)
F490 (= (INC-AMPS (E Q3)) 8.818342E-4) (F489 INC-KCL)
F489 (= (INC-AMPS (#1 R6)) -8.818342E-4) (F488 F464 F112 INC-OHM)
F488 (= (INC-VOLTS (#1 R6)) -0.55555555) (F487 INC-KVL)
F487 (= (INC-VOLTS (E Q3)) -0.55555555) (F484 F271 INC-BJT-ACTIVE)
F484 (= (INC-VOLTS (B Q3)) -0.55555555) (F483 INC-KVL)
F483 (= (INC-VOLTS (#2 R3)) -0.55555555) (F482 F446 F106 INC-OHM)
F482 (= (INC-AMPS (#1 R3)) 5.5555555E-5) (F481 INC-2T-R)
F481 (= (INC-AMPS (#2 R3)) -5.5555555E-5) (F480 F475 F472 INC-KCL)
F480 (= (INC-AMPS (C Q1)) 0.0) (F476 F474 INC-KCL-BJT)
F476 (= (INC-AMPS (B Q1)) 0.0) (F263 INC-BJT-BETA-INFINITE)
F475 (= (INC-AMPS (C Q2)) 5.5555555E-5) (F473 F461 INC-KCL-BJT)
F474 (= (INC-AMPS (E Q1)) 0.0) (F473 INC-KCL)
F473 (= (INC-AMPS (B Q2)) 0.0) (F266 INC-BJT-BETA-INFINITE)
F472 (= (INC-AMPS (B Q3)) 0.0) (F271 INC-BJT-BETA-INFINITE)
F471 (= (INC-AMPS (B Q4)) 0.0) (F274 INC-BJT-BETA-INFINITE)
F470 (= (INC-AMPS (E Q5)) 0.0) (F469 F468 INC-KCL-BJT)
F469 (= (INC-AMPS (C Q5)) 0.0) (F328 INC-BJT-CUTOFF-IC)
F468 (= (INC-AMPS (B Q5)) 0.0) (F328 INC-BJT-CUTOFF-IB)
F464 (= (INC-VOLTS (#2 R6)) 0.0) (F463 INC-KVL)
F463 (= (INC-VOLTS (#1 CB1)) 0.0) (F454 INC-CAP)
F461 (= (INC-AMPS (E Q2)) -5.5555555E-5) (F460 INC-KCL)
F460 (= (INC-AMPS (#1 R4)) 5.5555555E-5) (F456 F438 F108 INC-OHM)
F456 (= (INC-VOLTS (#2 R4)) 0.0) (F451 INC-KVL)
F454 (= (INC-VOLTS (#2 CB1)) 0.0) (F451 INC-KVL)
F453 (= (INC-VOLTS (#2 R13)) 0.0) (F451 INC-KVL)
F452 (= (INC-VOLTS (#2 R8)) 0.0) (F451 INC-KVL)
F451 (= (INC-VOLTS (#2 W146)) 0.0) (F450 INC-SHORT)
F450 (= (INC-VOLTS (#1 W146)) 0.0) (F229 INC-VPROP)
F446 (= (INC-VOLTS (#1 R3)) 0.0) (F442 INC-KVL)
F445 (= (INC-VOLTS (#1 R5)) 0.0) (F442 INC-KVL)
F444 (= (INC-VOLTS (#1 R12)) 0.0) (F442 INC-KVL)
F442 (= (INC-VOLTS (#2 W133)) 0.0) (F441 INC-SHORT)
F441 (= (INC-VOLTS (#1 W133)) 0.0) (F230 INC-VPROP)
F440 (= (INC-AMPS (#2 W214)) 0.0) (F439 INC-2T-SHORT)
F439 (= (INC-AMPS (#1 W214)) 0.0) (F231 INC-CPROP)
F438 (= (INC-VOLTS (#1 R4)) 0.1) (F437 INC-KVL)
F437 (= (INC-VOLTS (E Q2)) 0.1) (F436 F266 INC-BJT-ACTIVE)
F436 (= (INC-VOLTS (B Q2)) 0.1) (F435 INC-KVL)
```

```
F435 (= (INC-VOLTS (E Q1)) 0.1) (F432 F263 INC-BJT-ACTIVE)
F432 (= (INC-VOLTS (B Q1)) 0.1) (F431 INC-KVL)
F431 (= (INC-VOLTS (#2 CC1)) 0.1) (F430 INC-CAP)
F430 (= (INC-VOLTS (#1 CC1)) 0.1) (F232 INC-VPROP)
F328 (DETERMINED (MODE Q5) CUTOFF) (F327)
F327 (CHOICE (MODE Q5) CUTOFF) (GIVEN F257 F1 NEEDCHOICE)
F274 (DETERMINED (MODE Q4) BETA-INFINITE) (F273)
F273 (CHOICE (MODE Q4) BETA-INFINITE) (GIVEN F258 F1 NEEDCHOICE)
F271 (DETERMINED (MODE Q3) BETA-INFINITE) (F270)
F270 (CHOICE (MODE Q3) BETA-INFINITE) (GIVEN F259 F1 NEEDCHOICE)
F266 (DETERMINED (MODE Q2) BETA-INFINITE) (F265)
F265 (CHOICE (MODE Q2) BETA-INFINITE) (GIVEN F260 F1 NEEDCHOICE)
F263 (DETERMINED (MODE Q1) BETA-INFINITE) (F262)
F262 (CHOICE (MODE Q1) BETA-INFINITE) (GIVEN F261 F1 NEEDCHOICE)
F261 (NEEDCHOICE (MODE Q1)) (TRY-BJT)
F260 (NEEDCHOICE (MODE Q2)) (TRY-BJT)
F259 (NEEDCHOICE (MODE Q3)) (TRY-BJT)
F258 (NEEDCHOICE (MODE Q4)) (TRY-BJT)
F257 (NEEDCHOICE (MODE Q5)) (TRY-BJT)
F232 (= (INC-VOLTS (INPUT PROT2)) 0.1) (USER GIVEN)
F231 (= (INC-AMPS (OUTPUT PROT2)) 0.0) (USER GIVEN)
F230 (= (INC-VOLTS (+VCC PROT2)) 0.0) (USER GIVEN)
F229 (= (INC-VOLTS (GROUND PROT2)) 0.0) (USER GIVEN)
F126 (= (RESISTANCE R13) 82000.0) (GIVEN)
F124 (= (RESISTANCE R12) 39000.0) (GIVEN)
F122 (= (RESISTANCE R11) 470.0) (GIVEN)
F120 (= (RESISTANCE R10) 6800.0) (GIVEN)
F118 (= (RESISTANCE R9) 1000.0) (GIVEN)
F116 (= (RESISTANCE R8) 2200.0) (GIVEN)
F112 (= (RESISTANCE R6) 630.0) (GIVEN)
F110 (= (RESISTANCE R5) 5600.0) (GIVEN)
F108 (= (RESISTANCE R4) 1800.0) (GIVEN)
F106 (= (RESISTANCE R3) 10000.0) (GIVEN)
F1 (ALTERNATIVES (MODE BJT) (BETA-INFINITE CUTOFF SATURATED)) (GIVEN)
QED
```

We see that the precise resistances of resistors R3, R4, R5, R6, R8, R9, R10, R11, R12 and
R13 had an effect on the derivation of fact F543. Changing the resistance of any of the other
resistors (R1, R2, R7) will leave F543 still true. This can be understood by examining the path
along which F543 was deduced:

```
==> (explain 'f503)
F503 (= (INC-VOLTS (OUTPUT PROT2)) X495) (F502 INC-VPROP)
F502 (= (INC-VOLTS (#1 W214)) X495) (F501 INC-SHORT)
```

```
F501 (= (INC-VOLTS (#2 W214)) X495) (F497 INC-KVL)
F497 (= (INC-VOLTS (#1 R8)); X495) (GIVEN F327 F273 F270 F265 F262)
F327 (CHOICE (MODE Q5) CUTOFF) (GIVEN F257 F1 NEEDCHOICE)
F273 (CHOICE (MODE Q4) BETA-INFINITE) (GIVEN F258 F1 NEEDCHOICE)
F270 (CHOICE (MODE Q3) BETA-INFINITE) (GIVEN F259 F1 NEEDCHOICE)
F265 (CHOICE (MODE Q2) BETA-INFINITE) (GIVEN F260 F1 NEEDCHOICE)
F262 (CHOICE (MODE Q1) BETA-INFINITE) (GIVEN F261 F1 NEEDCHOICE)
F261 (NEEDCHOICE (MODE Q1)) (TRY-BJT)
F260 (NEEDCHOICE (MODE Q2)) (TRY-BJT)
F259 (NEEDCHOICE (MODE Q3)) (TRY-BJT)
F258 (NEEDCHOICE (MODE Q4)) (TRY-BJT)
F257 (NEEDCHOICE (MODE Q5)) (TRY-BJT)
F1 (ALTERNATIVES (MODE BJT) (BETA-INFINITE CUTOFF SATURATED)) (GIVEN)
QED
```

Since F503 depends on no additional resistance values, the circuit gain itself depends only on R3, R4, R5, R6, R8, R9 R10, R11, R12, and R13. To change the gain to 40, we could pick one of them and try different values for it until we found one producing a satisfactory value for the gain. But there is an easier way! We can give one of the resistors a *variable* value, and let it be determined in terms of the desired gain. First we tell the system to forget the value it already knows for one of the relevant resistors (R5); then we give it a new value which is a symbolic unknown (RR5).

In addition, we know that if we change the resistance of one of the *other* resistors, R1, R2 or R7, the circuit's gain will not be affected (as long as we don't change the resistance so much that the transistors move to different operating regions! But if that happens, we will see it). We can thus use those resistors to adjust other things about the circuit, after arriving at the desired gain.

```
==> (change '(resistance r5))
409 facts to check out
382 active facts left
NIL
```

The value of R5's resistance is now "forgotten" (actually, merely disbelieved), and so is everything that was deduced from it. The information that is left when R5's resistance is gone is:

+VCC

15V

INPUT
3.1V

CC1

R1   R3   R5   R12

3.3±.1   Q1   3.97.55V   Q3   CC2   10.2V   Q4

2.6±.1   Q2   3.3±.55V   R10   8.2V   9.6V

R2   2.0±.1 V   R6   Q5   R9   R11

R4   R7   R13   8.0V   OUTPUT

2.8V   CB1   R8

GROUND
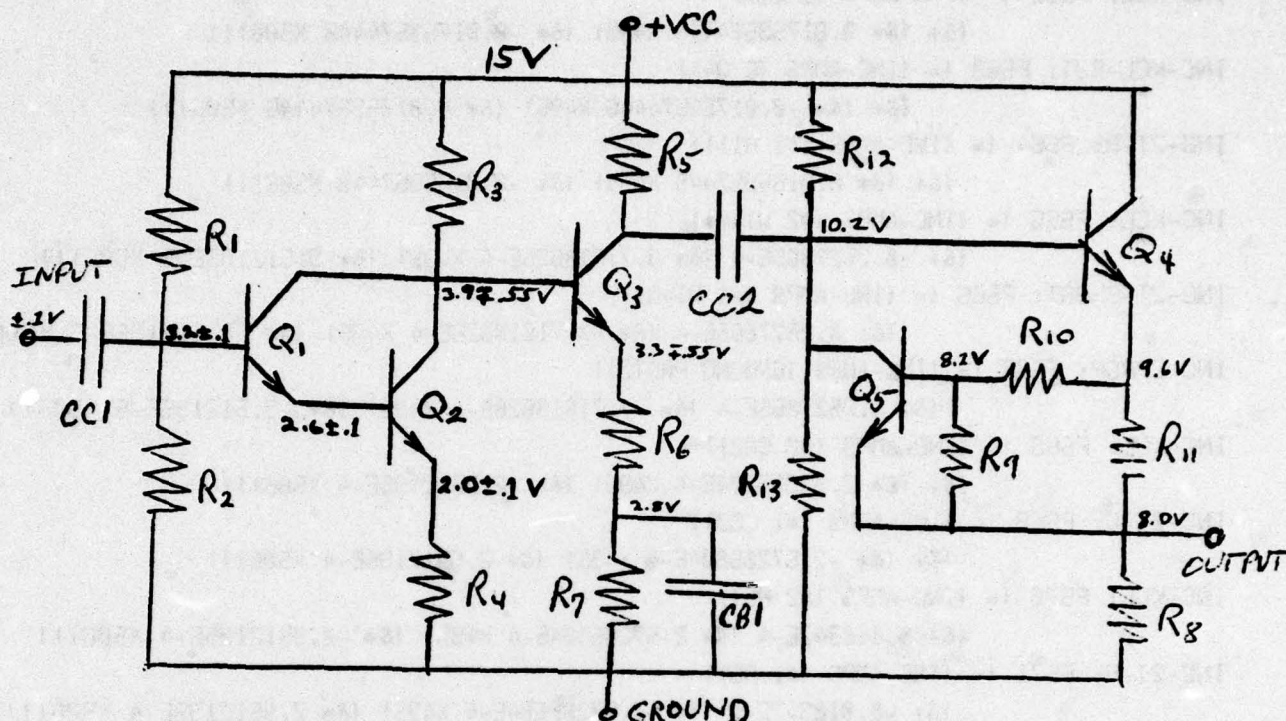
```
--> (tell '(resistance r5) 'rr5)
USER: F550 (= (RESISTANCE R5) RR5)
NIL
```

Now we must propagate the newly created unknown, RR5. This means determining the values of the circuit voltages and currents *as expressions involving that unknown.*

```
--> (run)
DC-OHM: F551 (= (VOLTAGE (#2 R5)) (&+ 15.0 (&* -8.488982E-4 RR5)))
DC-KVL: F552 (= (VOLTAGE (#1 CC2)) (&+ 15.0 (&* -8.488982E-4 RR5)))
DC-KVL: F553 (= (VOLTAGE (C Q3)) (&+ 15.0 (&* -8.488982E-4 RR5)))
VCB-MONITOR-BJT: F554 (HANGING D70 Q3 RR5)
VCB-MONITOR-BJT: F390 (HANGING D70 Q3 X369)
INC-2T-R: F555 (= (INC-AMPS (#2 R9)) (&+ (&* 1.0E-3 X495) (&* -1.0E-3 X506)))
INC-OHM: F556 (= (INC-AMPS (#1 R13)) (&+ (&* -8.292683E-5 X495) (&* 9.512195E-5 X506)))
INC-2T-R: F557 (= (INC-AMPS (#2 R13)) (&+ (&* 8.292683E-5 X495) (&* -9.512195E-5 X506)))
INC-OHM: F558 (= (INC-AMPS (#1 R11))
                (&+ (&* -0.0165957445 X495) (&* 0.0165957445 X506)))
```

INC-2T-R: F560 (CHECKED CHECK559 D4 R9)
INC-2T-R: F561 (= (INC-AMPS (#2 R10)) (&+ (&* -1.0E-3 X495) (&* 1.0E-3 X506)))
INC-KCL: F562 (= (INC-AMPS (E Q4))
                    (&+ (&* 0.01759574446 X495) (&* -0.01759574446 X506)))
INC-KCL-BJT: F563 (= (INC-AMPS (C Q4))
                        (&+ (&* -0.01759574446 X495) (&* 0.01759574446 X506)))
INC-2T-R: .F564 (= (INC-AMPS (#2 R11))
                    (&+ (&* 0.0165957445 X495) (&* -0.0165957445 X506)))
INC-KCL: F565 (= (INC-AMPS (#2 W146))
                    (&+ -8.2527865E-4 (&* 3.71618626E-4 X495) (&* 9.512195E-5 X506)))
INC-2T-SHORT: F566 (= (INC-AMPS (#1 W146))
                        (&+ 8.2527865E-4 (&* -3.71618626E-4 X495) (&* -9.512195E-5 X506)))
INC-CPROP: F567 (= (INC-AMPS (GROUND PROT2))
                        (&+ 8.2527865E-4 (&* -3.71618626E-4 X495) (&* -9.512195E-5 X506)))
INC-KCL: F568 (= (INC-AMPS (#2 CC2))
                        (&+ (&* 2.57285804E-4 X495) (&* -2.9512195E-4 X506)))
INC-2T-C: F569 (= (INC-AMPS (#1 CC2))
                        (&+ (&* -2.57285804E-4 X495) (&* 2.9512195E-4 X506)))
INC-KCL: F570 (= (INC-AMPS (#2 R5))
                        (&+ 8.818342E-4 (&* 2.57285804E-4 X495) (&* -2.9512195E-4 X506)))
INC-2T-R: F571 (= (INC-AMPS (#1 R5))
                        (&+ -8.818342E-4 (&* -2.57285804E-4 X495) (&* 2.9512195E-4 X506)))
INC-KCL: F572 (= (INC-AMPS (#2 W133))
                        (&+ 8.26548922E-4 (&* 0.0176786713 X495) (&* -0.0176908665 X506)))
INC-2T-SHORT: F573 (= (INC-AMPS (#1 W133))
                        (&+ -8.26548922E-4
                            (&* -0.0176786713 X495)
                            (&* 0.0176908665 X506)))
INC-CPROP: F574 (= (INC-AMPS (+VCC PROT2))
                        (&+ -8.26548922E-4 (&* -0.0176786713 X495) (&* 0.0176908665 X506)))
INC-OHM: F576 (CHECKED CHECK575 D3 R5)
INC-OHM: F577 (VALUE RR5 (&// (&+ (&* 6.8 X495) (&* -7.8 X506))
                                (&+ -8.818342E-4
                                    (&* -2.57285804E-4 X495)
                                    (&* 2.9512195E-4 X506))))
VCB-MONITOR-BJT: F578 (HANGING D70 Q3 X506)
VCB-MONITOR-BJT: F579 (HANGING D70 Q3 X495)
INC-2T-R: F581 (CHECKED CHECK580 D4 R13)
INC-KCL: F582 (VALUE X506 (&* 1.0258327 X495))
INC-KCL: F584 (CHECKED CHECK583 D36 N219)
DONE

The value of RR5, and thus of R5's resistance, is known only in terms of another

unknown, X495, because, with one less quantity given than before, there is no longer enough information to determine all the circuit parameters. We can remedy that by specifying the output signal level as 4.0 volts, which is a gain of 40 over the input we have supplied:

```
==> (tell '(inc-volts (output prot2)) 4.0)
USER: F585 (= (INC-VOLTS (OUTPUT PROT2)) 4.0)
NIL
==> (run)
VCB-MONITOR-BJT: F586 (VALUE X495 4.0)
INC-VPROP: F588 (CHECKED CHECK587 D44 TERMINAL216)
DONE
```

           The appropriate value of R5's resistance is now known.

```
==> (what '(resistance r5))
6865.7402 (F582 F586 F577 F550)
T
==> (what '(inc-volts (#2 r5)))
4.80598 (F582 F586 F523)
T
==> (what '(voltage (#2 r5)))
9.17168545 (F582 F586 F577 F551)
T
==> (what '(voltage (e q3)))
3.33617 (F423 F388)
T
==> (what '(inc-volts (e q3)))
-0.55555555 (F487)
T
```

But, by changing the value of R5 to get the right gain, we have also altered the biasing of the circuit. We now adjust the bias to the desired level by changing R7, which EL has told us *will not affect the gain*.

```
==> (change '(resistance r7))
417 facts to check out
351 active facts left
NIL
==> (what '(resistance r5))
6865.7402 (F582 F586 F577 F550)
T
==> (what '(voltage (c q3)))
?
```

```
==> (tell '(voltage (c q3)) 9.5)
USER: F589 (= (VOLTAGE (C Q3)) 9.5)
NIL
==> (run)
DC-KVL: F590 (= (VOLTAGE (#1 CC2)) 9.5)
DC-KVL: F591 (= (VOLTAGE (#2 R5)) 9.5)
DC-OHM: F592 (= (CURRENT (#1 R5)) 8.01078963E-4)
DC-2T-R: F593 (= (CURRENT (#2 R5)) -8.01078963E-4)
DC-KCL: F594 (= (CURRENT (C Q3)) 8.01078963E-4)
DC-KCL-BJT: F595 (= (CURRENT (E Q3)) -8.01078963E-4)
DC-KCL: F596 (= (CURRENT (#1 R6)) 8.01078963E-4)
DC-OHM: F597 (= (VOLTAGE (#1 R6)) (&+ 0.50467974 X369))
DC-KVL: F598 (= (VOLTAGE (E Q3)) (&+ 0.50467974 X369))
DC-BJT-ACTIVE: F599 (= (VOLTAGE (B Q3)) (&+ 1.10467975 X369))
VCB-MONITOR-BJT: F390 (HANGING D70 Q3 X369)
VBE-MONITOR-BJT: F391 (HANGING D71 Q3 X369)
DC-KVL: F600 (= (VOLTAGE (C Q2)) (&+ 1.10467975 X369))
DC-KVL: F601 (= (VOLTAGE (C Q1)) (&+ 1.10467975 X369))
DC-KVL: F602 (= (VOLTAGE (#2 R3)) (&+ 1.10467975 X369))
DC-OHM: F603 (= (CURRENT (#1 R3)) (&+ 1.38953203E-3 (&* -1.0E-4 X369)))
DC-2T-R: F604 (= (CURRENT (#2 R3)) (&+ -1.38953203E-3 (&* 1.0E-4 X369)))
DC-KCL: F605 (= (CURRENT (C Q2)) (&+ 1.38953203E-3 (&* -1.0E-4 X369)))
IC-MONITOR-BJT: F398 (HANGING D66 Q2 X369)
NOTE-SATURATED: F399 (HANGING D72 Q2 X369)
DC-KCL-BJT: F606 (= (CURRENT (E Q2)) (&+ -1.38953203E-3 (&* 1.0E-4 X369)))
IE-MONITOR-BJT: F401 (HANGING D65 Q2 X369)
DC-KCL: F607 (= (CURRENT (#1 R4)) (&+ 1.38953203E-3 (&* -1.0E-4 X369)))
DC-OHM: F608 (= (VOLTAGE (#1 R4)) (&+ 2.50115764 (&* -0.18 X369)))
DC-KVL: F609 (= (VOLTAGE (E Q2)) (&+ 2.50115764 (&* -0.18 X369)))
DC-BJT-ACTIVE: F610 (= (VOLTAGE (B Q2)) (&+ 3.10115764 (&* -0.18 X369)))
VCB-MONITOR-BJT: F406 (HANGING D70 Q2 X369)
VBE-MONITOR-BJT: F407 (HANGING D71 Q2 X369)
DC-KVL: F611 (= (VOLTAGE (E Q1)) (&+ 3.10115764 (&* -0.18 X369)))
DC-BJT-ACTIVE: F612 (= (VOLTAGE (B Q1)) (&+ 3.7011576 (&* -0.18 X369)))
VCB-MONITOR-BJT: F410 (HANGING D70 Q1 X369)
VBE-MONITOR-BJT: F411 (HANGING D71 Q1 X369)
DC-KVL: F613 (= (VOLTAGE (#1 R2)) (&+ 3.7011576 (&* -0.18 X369)))
DC-KVL: F614 (= (VOLTAGE (#2 R1)) (&+ 3.7011576 (&* -0.18 X369)))
DC-KVL: F615 (= (VOLTAGE (#2 CC1)) (&+ 3.7011576 (&* -0.18 X369)))
DC-OHM: F616 (= (CURRENT (#1 R1)) (&+ 3.0537412E-5 (&* 4.8648648E-7 X369)))
DC-KCL: F617 (= (CURRENT (#2 W133)) (&+ -5.9638228E-3 (&* 9.9513513E-5 X369)))
DC-2T-SHORT: F618 (= (CURRENT (#1 W133)) (&+ 5.9638228E-3 (&* -9.9513513E-5 X369)))
DC-CPROP: F619 (= (CURRENT (+VCC PROT2)) (&+ 5.9638228E-3 (&* -9.9513513E-5 X369)))
```

```
DC-2T-R: F620 (= (CURRENT (#2 R1)) (&+ -3.0537412E-5 (&* -4.8648648E-7 X369)))
DC-KCL: F621 (= (CURRENT (#1 R2)) (&+ 3.0537412E-5 (&* 4.8648648E-7 X369)))
DC-OHM: F623 (CHECKED CHECK622 D1 R2)
DC-OHM: F624 (VALUE X369 2.83149037)
DC-2T-R: F424 (= (CURRENT (#2 R2)) -3.1914894E-5)
DC-2T-R: F425 (= (CURRENT (#2 R4)) -1.106383E-3)
DC-2T-R: F625 (= (CURRENT (#2 R6)) -8.01078963E-4)
DC-KCL: F626 (= (CURRENT (#1 R7)) 8.01078963E-4)
DC-OHM: F627 (= (RESISTANCE R7) 3534.5958)
DC-2T-R: F628 (= (CURRENT (#2 R7)) -8.01078963E-4)
DC-KCL: F629 (= (CURRENT (#2 W146)) 5.6820512E-3)
DC-2T-SHORT: F630 (= (CURRENT (#1 W146)) -5.6820512E-3)
DC-CPROP: F631 (= (CURRENT (GROUND PROT2)) -5.6820512E-3)
INC-OHM: F466 (= (INC-AMPS (#1 R7)) 0.0)
    Unout: F493 (= (INC-AMPS (#1 CB1)) -8.818342E-4)
    Unout: F494 (= (INC-AMPS (#2 CB1)) 8.818342E-4)
    Unout: F467 (= (INC-AMPS (#2 R7)) 0.0)
    Unout: F565 (= (INC-AMPS (#2 W146))
                (&+ -8.2527865E-4 (&* 3.71618626E-4 X495) (&* 9.512195E-5 X506)))
    Unout: F566 (= (INC-AMPS (#1 W146))
                (&+ 8.2527865E-4 (&* -3.71618626E-4 X495) (&* -9.512195E-5 X506)))
    Unout: F567 (= (INC-AMPS (GROUND PROT2))
                (&+ 8.2527865E-4 (&* -3.71618626E-4 X495) (&* -9.512195E-5 X506)))
INC-KCL: F633 (CHECKED CHECK632 D36 N151)
DONE
```

Now we have determined the value of R7 that we need to get the desired operating point.
The amplifier's design is now complete; after asking for the component values we have just
determined, we are ready to build the circuit and verify our results.

```
==> (what '(resistance r5))
6865.7402 (F582 F586 F577 F550)
T
==> (what '(resistance r7))
3534.5958 (F627)
T
==> (what '(voltage (e q3)))
3.3361701 (F624 F598)
T
==> (what '(inc-volts (e q3)))
-0.55555555 (F487)
T
```

Notes

Blind alleys

Various other systems have been investigated which try to make effective use of the results of blind alleys, including:

BUILD <Fahlman 1973>, TOPLE <McDermott 1974>, HACKER <Sussman 1973,1975>

Limiters

One way to prevent combinatorial explosions is to formulate one's domain so that there is a finite set of "slots" to be filled. Nevins' geometry theorem prover <Nevins 1974> makes use of this property of geometry problems without constructions: "Although the program does not use a diagram directly, it does take advantage of the manageable problem space implicit in the diagram. An important aspect of human problem solving may be the ability to structure a problem space so that forward chaining techniques can be used effectively." The FRAME systems approach of Minsky <Minsky 1974> and Winograd <Winograd 1974> shares this essential finiteness of the data structure.

Complexity

One of the major limitations on the development of large expert programs is the complexity barrier encountered when the program gets so large that it is hard to keep the interactions between its parts under control. Various solutions to this problem have been offered. The "structured programming" movement <Dijkstra 1970> proposes to constrain the style of programmers so that the interactions are clarified and limited. Others <Teitelman 1970> <Winograd 1973> propose to build systems which supply stronger support to programmers in terms of routine bookkeeping. The extreme form of this idea leads to various "automatic programming" efforts. These range from "programming apprentices" <Hewitt & Smith 1975>, <Rich & Shrobe 1976> to expert system compilers <AUTOPROG 1975> and various automatic synthesis systems <Sussman 1973,1975> <Sussman 1974>. Another approach is to try to constrain the representation of knowledge in a program to be as simple and modular as possible. This has led to various attempts to build "rule-based" systems and "advice takers" such as MYCIN <Shortliffe 1974> <Davis 1976> and EL. NASL <McDermott 1976> is a computer programming language which attempts to provide features for organizing programs along these lines.

Explainers

In the MYCIN system <Shortliffe 1974> <Davis 1976> the ability to generate coherent explanations of its reasoning plays an important part in the debugging of the rule set and in the acceptance of its conclusions by its users.

LISP

The ARS program is implemented in MACLISP <Moon 1974>, a version of LISP <McCarthy 1965>, which was created and is used at the MIT Artificial Intelligence Laboratory and MIT Project MAC. LISP has been the language of many large and interesting programs because of its elegance, simplicity, and convenience for symbolic manipulation. MACLISP is available on

the MULTICS, TOPS-10, and the Incompatible Timesharing systems.

ARS
Antecedent Reasoning System

Pattern-directed invocation

Pattern-directed invocation <Hewitt 1971> is an artificial intelligence programming technique whereby a program can be executed without knowing its name. There are two forms of pattern-directed invocation. In one kind, a routine definition specifies a pattern which "advertises" the kind of problem the routine is useful for. The routine can then be invoked by a special kind of call which specifies a pattern describing the problem to be solved. This kind of pattern-directed invocation may be used to implement consequent reasoning. In the kind we use, a "demon" program may be defined with a pattern which specifies that it should be invoked whenever an event matching that pattern occurs. This kind of demon is useful for monitoring an indexed data base. It may be awakened by any change in the data base matching its pattern of invocation. This kind of pattern-directed invocation may be used to implement antecedent reasoning.

Data Bases

An important feature of most AI languages (e.g. Planner <Hewitt 1972>, Micro-Planner <Sussman, Winograd & Charniak 1970>, Conniver <McDermott & Sussman 1972>, QA4 <Rulifson 1972>) is the indexed data base. It is often important to be able to record a "fact" in so that it can be accessed in many ways. For example, we may want to record the information that Bl is a big red block so that if later a program wants to know what blocks are red or if there are any big red objects, the system can retrieve that information even though the asserting process does not know what questions are going to be asked. The indexed data base is fundamentally a fully inverted file of records of variable length. Efficient means of implementing such structures are still a matter of research interest <McDermott 1975>.

Context

TOPLE <McDermott 1974> was an early attempt to record the interactions among deductions for the purpose of deciding what is currently believed to be true. McDermott used this information to help decide which of several assumptions must be thrown out in order to keep a consistent data base when a new fact conflicted with existing ones. MYCIN <Shortliffe 1974> <Davis 1976> use dependency information to produce explanations but do not use it for any control purposes. The SRI Computer Based Consultant <Fikes 1975> makes use of dependencies to determine the logical support of facts in a manner similar to ARS but does not use them to control search.

EL

A previous version of EL, which was much more primitive than the one we describe here, was implemented directly in LISP <Sussman & Stallman 1975>.

Propagation

Analysis by propagation of constraints is also implemented in INTER <de Kleer 1976>. A similar process of relaxation of symbolic constraints has been applied to the labelling of line drawings of visual scenes <Huffman 1970>. A beautiful exposition of this technique can be found in <Waltz 1972>. Some theoretical analysis of this technique appears in <Freuder 1976>.

Symbolic manipulation

Much more powerful and extensive symbolic algebraic manipulation systems have been written than the one we use in EL. The most powerful one we know of is MACSYMA <MACSYMA 1974>

Advice

In a real design situation, the designer knows the intended operating point region of the devices in question. The user of EL may, of course, supply any information which EL could deduce as "Advice" to the system. If a contradiction occurs due to his advice, he should be very interested!

Backtracking

Chronological backtrack control was implemented in Micro-Planner <Sussman, Winograd & Charniak 1970>. Users of Micro-Planner were plagued by serious problems <Sussman & McDermott 1972>. The control of the combinatorial searches performed by Micro-Planner is nearly impossible due to the large number of anomalous dependencies (in ARS terms) introduced by the chronological component of the backtrack stack.

Braida

This measure of simultaneity was proposed by Louis D. Braida.

Derivation

A derivation of this interesting behavior can be found in <Senturia & Wedlock 1975> pp. 270-276

Queueless

NASL <McDermott 1976> is implemented with a similar queueless scheme for interpreting rules. This scheme is very elegant and flexible.

Equality

Lists which are EQUAL in the LISP sense have the same properties, so Interlisp's <INTERLISP 1974> hash-links would *not* fill the bill.

Discrimination nets

The decision tree method of implementing a pattern data-base for demons is similar to the scheme used in QA4 <Rulifson 1972>.

Bibliography

<AUTOPROG 1975>

Bosyj, Michael <u>A diagnostic program for the design of procurement systems</u> MIT Project
MAC Technical Report *** (July 1975)

<Davis 1976>

Davis, Randall <u>Applications of meta level knowledge to the construction, maintenance,
and use of large knowledge bases</u> Stanford U. AI Lab. Memo AIM-283 (July 1976)

<de Kleer 1976>

de Kleer, Johan <u>Local Methods for Localization of Faults in Electronic Circuits</u> MIT AI
Lab. Working Paper 109 (September 1976)

<Dijkstra 1970>

Dijkstra, Edsgar W. "Structured Programming" in <u>Software engineering techniques,</u>
J.N.Buxton & B.Randell(Eds.) NATO Scientific Affairs Division, Brussels, Belgium, 1970, 84-88.

<Doyle 1976>

Doyle, Jon. "Analysis by Propagation of Constraints in Elementary Geometry Problem
Solving" MIT AI Lab. Working Paper 108 (April 1976)

<Fahlman 1973>

Fahlman, Scott Elliot <u>A Planning System for Robot Construction Tasks</u> MIT Dept. of
EE&CS MS Thesis, MIT AI Lab Technical Report 283 (May 1973)

<Fikes 1975>

Fikes, Richard E. <u>Deductive Retrieval Mechanisms for State Description Models</u>
Stanford Research Institute AI Technical Note 106 (July 1975)

<Freuder 1970>

Freuder, Eugene C. <u>Synthesizing Constraint Expressions</u> MIT AI Lab. Memo 370 (July
1976)

<Hewitt & Smith 1975>

Hewitt, Carl E. & Smith, Brian "Towards a Programming Apprentice" in <u>IEEE Trans.
on Software Engineering</u> Vol.1 No.1 pp.26-45 (March 1975)

<Hewitt 1971>

Hewitt, Carl E. "Procedural Embedding of Knowledge in PLANNER" in <u>Proc. IJCAI 2</u>
(September 1971)

<Hewitt 1972>

     Hewitt, Carl E. Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot MIT AI Lab. Technical Report 258 (April 1972)

<Huffman 1970>

     Huffman, David A. "Impossible Objects as Nonsense Sentences" in Machine Intelligence 6 Edinburgh, U.K.: Edinburgh University Press (1970)

<INTERLISP 1974>

     Teitelman, Warren et al. INTERLISP Reference Manual XEROX Palo Alto Research Center and Bolt Beranek and Newman (1974)

<MACSYMA 1974>

     Bogen, Richard et al. MACSYMA Reference Manual MIT Project MAC (September 1974)

<Mason 1976>

     Mason, Matthew T. Qualitative Simulation of Swine Production MIT Dept. of EE&CS BS Thesis (May 1976)

<McCarthy 1965>

     McCarthy et al. LISP 1.5 Programmer's Manual Cambridge Mass.: M.I.T. Press (1965)

<McDermott & Sussman 1972>

     McDermott, Drew Vincent and Sussman, Gerald Jay CONNIVER Reference Manual MIT AI Lab. AI Memo 259 (May 1972, revised July 1973)

<McDermott 1974>

     McDermott, Drew Vincent Assimilation of New Information by a Natural Language Understanding System MIT Dept. of EE&CS MS Thesis, MIT AI Lab Technical Report 291 (February 1974)

<McDermott 1975>

     McDermott, Drew Vincent Very Large Planner-type Data Bases MIT AI Lab. Memo 339 (1975)

<McDermott 1976>

     McDermott, Drew Vincent Flexibility and Efficiency in a Computer Program for Designing Circuits MIT Dept. EE&CS PhD Thesis (September 1976)

<Minsky 1974>

     Minsky, Marvin Lee "A Framework for Representing Knowledge" MIT AI Lab Memo

306 (June 1974), in The Psychology of Computer Vision P.H.Winston McGraw-Hill (1975)

<Moon 1974>

        Moon, David A. MACLISP Reference Manual MIT Project MAC (April 1974)

<Nevins 1974>

        Nevins, Arthur J. Plane Geometry Theorem Proving Using Forward Chaining MIT
Artificial Intelligence Memo 303 (January 1974)

<Rich & Shrobe 1976>

        Rich, Charles & Shrobe, Howie Initial Report on a LISP Programmer's Apprentice MIT
AI Lab. Technical Report 354 (Sept. 1976)

<Roylance 1975>

        Roylance, Gerald Lafael Anthropomorphic Circuit Analysis MIT Dept. of EE&CS BS
Thesis (June 1975)

<Roylance 1976>

        Roylance, Gerald Lafael Qualitative Analysis of Operational Amplifier Circuits MIT
Dept. of EE&CS MS Thesis proposal (March 1976)

<Rulifson 1972>

        Rulifson, Johns F. et al. A procedural calculus for intuitive reasoning Stanford Research
Institute AI Technical Note 73 (November 1972)

<Senturia & Wedlock 1975>

        Senturia, Stephen D. and Wedlock, Bruce D. Electronic Circuits and Applications New
York: John Wiley and Sons (1975)

<Shortliffe 1974>

        Shortliffe, E.H. MYCIN -- A rule-based computer program for advising physicians
regarding antimicrobial therapy selection Stanford U. AI Lab. Memo AIM-251 (October 1974), also
MYCIN: Computer-Based Medical Consultations, New York: Elsevier (1976)

<Sussman & McDermott 1972>

        Sussman, Gerald Jay and McDermott Drew Vincent "From PLANNER to CONNIVER
-- A genetic approach" in Proc. FJCC (1972) pp. 1171-1179

<Sussman & Stallman 1975>

        Sussman, Gerald Jay and Stallman, Richard Matthew "Heuristic Techniques in
Computer-Aided Circuit Analysis" in IEEE Trans. on Circuits and Systems vol. CAS-22 No. 11
(November 1975)

<Sussman 1973,1975>

Sussman, Gerald Jay <u>A Computer Model of Skill Acquisition</u> MIT Dept of Math. PhD Thesis (August 1973), also New York: Elsevier (1975)

<Sussman 1974>

Sussman, Gerald Jay "The Virtuous Nature of Bugs" in <u>Proc. of the AISB conference</u> Sussex, U.K. (July 1974)

<Sussman, Winograd & Charniak 1970>

Sussman, Gerald Jay, Winograd, Terry, and Charniak, Eugene <u>Micro-Planner Reference Manual</u> MIT AI Lab. AI Memo 203 (July 1970, revised December 1971)

<Teitelman 1970>

Teitelman, Warren. "Toward a Programming Laboratory" in <u>Software engineering techniques,</u> J.N.Buxton & B.Randell(Eds.) NATO Scientific Affairs Division, Brussels, Belgium, 1970, 137-149

<Waltz 1972>

Waltz, David L. <u>Generating Semantic Descriptions from Drawings of Scenes with Shadows</u> MIT AI Lab. Technical Report 271 (November 1972)

<Winograd 1973>

Winograd, Terry "Breaking the Complexity Barrier Again" in <u>Proc. ACM SIGPN/SIGIR Interface Meeting</u> (November 1973)

<Winograd 1974>

Winograd, Terry "Frame Representations and the Procedural/Declarative Controversy" in <u>Representation & Understanding,</u> D.G.Bobrow & A.Collins (Eds.) New York: Academic Press (1975)